

AD-A194 886

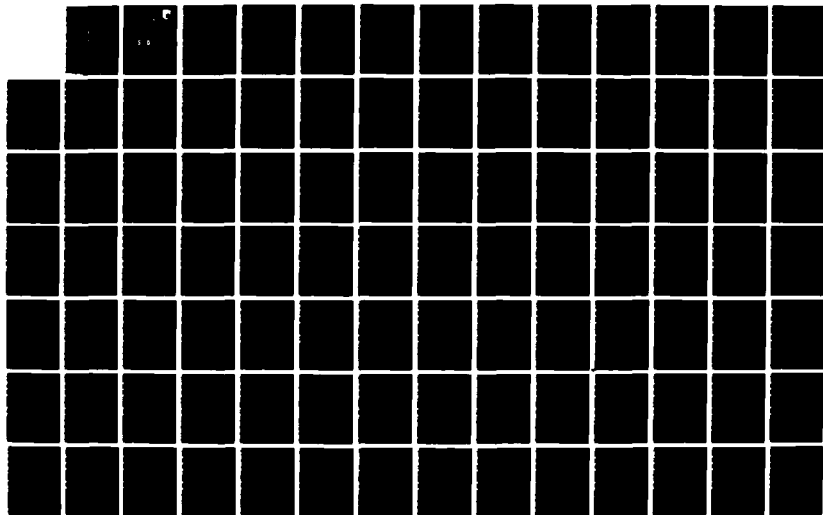
A MULTIPROCESSOR AVIONICS SYSTEM FOR AN UNMANNED
RESEARCH VEHICLE(U) AIR FORCE WRIGHT AERONAUTICAL LABS
WRIGHT-PATTERSON AFB OH D B THOMPSON MAR 88
AFWAL-TR-88-3003

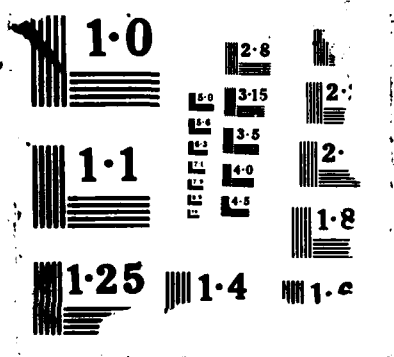
1/3

UNCLASSIFIED

F/G 12/6

NL

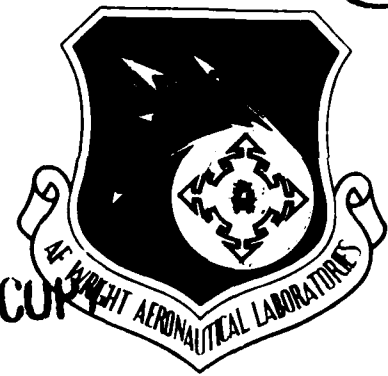




AFWAL-TR-88-3003

nor
A MULTIPROCESS AVIONICS SYSTEM FOR AN UNMANNED RESEARCH
VEHICLE

DTIC FILE COPY



Daniel B. Thompson
Control Systems Development Branch
Flight Control Division

March 1988

AD-A194 806

Final Report for Period January 1987 - December 1987

DTIC
ELECTE
MAY 18 1988
S D

Approved for Public Release; Distribution is Unlimited

FLIGHT DYNAMICS LABORATORY
AIR FORCE WRIGHT AERONAUTICAL LABORATORIES
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-6553

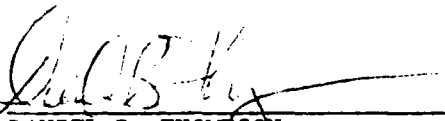
88 5 17 018

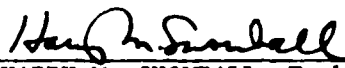
NOTICE

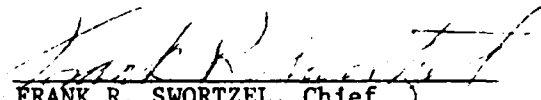
When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the Government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise as in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report has been reviewed by the Office of Public Affairs (ASP/PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

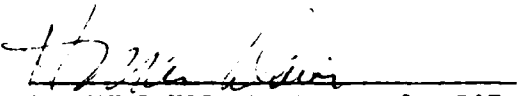
This technical report has been reviewed and is approved for publication.


DANIEL B. THOMPSON
Project Manager
Control Systems Development Branch
Flight Control Division


HARRY M. SNOWBALL, Tech. Grp. Mgr.
Control Data Group
Control Systems Development Branch
Flight Control Division


FRANK R. SWORTZEL, Chief)
Control Systems Development Branch
Flight Control Division

FOR THE COMMANDER


H. MAX DAVIS, Assistant for R&T
Flight Control Division
Flight Dynamics Laboratory

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify AFWAL/FICL, Wright-Patterson AFB, OH 45433-6553 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

REPORT DOCUMENTATION PAGE

| | | | | |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------|-----------------------------------------------------|----------------------------------------------------------------------------------------------------------------------|-----------------------------------|
| 1a. REPORT SECURITY CLASSIFICATION Unclassified | | | 1b. RESTRICTIVE MARKINGS | |
| 2a. SECURITY CLASSIFICATION AUTHORITY | | | 3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release; Distribution is Unlimited | |
| 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE | | | 4. PERFORMING ORGANIZATION REPORT NUMBER(S) | |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S) | | | 5. MONITORING ORGANIZATION REPORT NUMBER(S) AFWAL-TR-88-3003 | |
| 6a. NAME OF PERFORMING ORGANIZATION Flight Dynamics Laboratory | 6b. OFFICE SYMBOL (if applicable) AFWAL/FI GL | 7a. NAME OF MONITORING ORGANIZATION | | |
| 6c. ADDRESS (City, State, and ZIP Code) Flight Dynamics Laboratory (AFWAL/FI GL) AF Wright Aeronautical Laboratories (AFSC) Wright-Patterson AFB OH 45433 | | 7b. ADDRESS (City, State, and ZIP Code) | | |
| 8a. NAME OF FUNDING/SPONSORING ORGANIZATION | 8b. OFFICE SYMBOL (if applicable) | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER | | |
| 8c. ADDRESS (City, State, and ZIP Code) | | 10. SOURCE OF FUNDING NUMBERS | | |
| | | PROGRAM ELEMENT NO. 62201F | PROJECT NO. 2403 | TASK NO. 02 |
| | | WORK UNIT ACCESSION NO. 99 | | |
| 11. TITLE (Include Security Classification) A Multiprocessor Avionics System for an Unmanned Research Vehicle | | | | |
| 12. PERSONAL AUTHOR(S) Daniel B. Thompson | | | | |
| 13a. TYPE OF REPORT Final | 13b. TIME COVERED FROM 1 Jan 87 to 15 Dec 87 | 14. DATE OF REPORT (Year, Month, Day) 1988 March | 15. PAGE COUNT 197 | |
| 16. SUPPLEMENTARY NOTATION Thesis results contributing to in-house multiprocessor and URV research | | | | |
| 17. COSATI CODES | | | 18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) | |
| FIELD | GROUP | SUB-GROUP | Multiprocessor, Unmanned Research Vehicle, Flight Control, Operating Systems, Microprocessors, Software Development. | |
| 09 | 02 | | | |
| 23 | 06 | | | |
| 19. ABSTRACT (Continue on reverse if necessary and identify by block number) AFWAL/FI GL is developing a new Unmanned Research Vehicle (URV) testbed system for low cost flight testing of advanced flight control concepts. This new system will incorporate a modular/reconfigurable airframe, will possess greater aerodynamic capabilities, and will utilize an advanced avionics/control system to allow embedded computation of flight test applications. AFWAL/FI GL has utilized its in-house experience in multiprocessor systems technologies to develop a first phase prototype system comprised of multiple microprocessors and a parallel backplane bus. A multiprocessor software operating system and interprocessor communications protocol have been developed and tested. To demonstrate the capabilities of the system and the development of parallel software, an applications set of parallel software tasks have been developed and demonstrated. The hardware and software architecture, approach taken in the development, and results achieved are described in this report. | | | | |
| 20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT <input type="checkbox"/> DTIC USERS | | | 21. ABSTRACT SECURITY CLASSIFICATION Unclassified | |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL Daniel B. Thompson | | | 22b. TELEPHONE (Include Area Code) (513) 255-8288 | 22c. OFFICE SYMBOL AFWAL/FI GL |

ABSTRACT

Thompson, Daniel B. M.S., Department of Computer Science, Wright State University, 1987. A Multiprocessor Avionics System For An Unmanned Research Vehicle.

The Air Force Flight Dynamics Laboratory is developing a new Unmanned Research Vehicle (URV) to support low cost/risk in-house flight tests of advanced flight control concepts. Implicit to the development of the testbed is the addition of an advanced on-board avionics system to support computationally intensive embedded tasks. As the first phase of the development of this avionics system, a prototype multiprocessor system and operating system software have been designed and tested. As demonstration of its capabilities, the prototype implements a control mixer algorithm for control surface reconfiguration in the event of failure. The analysis, design, development, and test procedures and results of the research of the prototype are described. Areas of further research in the following phases of development are also discussed.



| | |
|--------------------|--------------------------------------------|
| Accession For | |
| NTIS CRA&I | <input checked="checked" type="checkbox"/> |
| DTIC TAB | <input type="checkbox"/> |
| Unannounced | <input type="checkbox"/> |
| Justification | |
| By | |
| Distribution/ | |
| Availability Codes | |
| Date | Avail and/or Special |
| A-1 | |

TABLE OF CONTENTS

| | Page |
|---------------------------------------------------------------------------------------------------|------|
| (1) Introduction | 1 |
| (2) Background | 4 |
| (3) Design Considerations of a Multiprocessor URV Avionics System | 9 |
| (4) Overall Goals to Architecture Specification . . | 12 |
| (4.1) Computation Area | 12 |
| (4.2) Interprocessor Communications Area | 13 |
| (4.3) I/O Area | 17 |
| (4.4) Resulting Configuration and Hardware Selection | 21 |
| (5) Software Specification | 25 |
| (5.1) Techniques of Multiprocessor Software | 25 |
| (5.2) Programmer Viewpoint: Tasks | 27 |
| (5.3) Analysis of Available Real Time Operating Systems (RTOS) | 28 |
| (5.4) Analysis of the Multiprocessor/Multitasking Problem in the Control Environment | 31 |
| (5.4.1) The Multiple Processor Problem | 31 |
| (5.4.2) Intertask Communications | 33 |
| (5.4.3) Control Timing | 34 |
| (5.5) URV Real Time Multiprocessor/multitasking Operating System (RTMOS) Kernel Specification | 35 |
| (5.5.1) Kernel Structures | 36 |
| (5.5.2) Task Data Exchanges and Context Switches . . | 38 |
| (5.6) Remaining RTMOS Features | 48 |
| (6) Laboratory Approach | 53 |
| (6.1) Processor Board | 53 |

TABLE OF CONTENTS (CONTINUED)

| | Page |
|-------------------------------------------------------------------------------|------|
| (6.2) Kernel Development | 54 |
| (6.3) Floating Point Hardware | 56 |
| (6.4) Applications Software Development | 57 |
| (6.5) Test Configuration | 61 |
| (6.6) Parallelization of the Ki Function | 61 |
| (6.7) Allocation of Tasks onto Multiple Processors . | 70 |
| (6.8) 8061 Hardware Design | 72 |
| (6.9) 8061 Software Modifications | 75 |
| (6.10) Hardware-in-the-Loop Simulation Configuration | 75 |
| (7) Prototype Development and Demonstration Results and Findings | 79 |
| (7.1) VME Performance and Bus Loading | 79 |
| (7.1.1) VME Access Options | 80 |
| (7.1.2) VME Bus Access Latency Effects (Theory) . . | 81 |
| (7.1.3) VME Bus Access Latency Effects (Measured) . | 82 |
| (7.1.4) VME Bus Access Latency Effects (Conclusions) | 83 |
| (7.2) Execution Times for Kernel Routines | 83 |
| (7.2.1) RTOS Comparisons | 84 |
| (7.2.2) RTMOS Timings | 84 |
| (7.2.3) Implications of the RTMOS Timing Data . . . | 86 |
| (7.3) Applications Computation Times | 87 |
| (7.3.1) Sequential Limitations | 87 |
| (7.3.2) Measured Computation Times | 88 |
| (7.3.3) Implications of the Execution Time Data . . | 90 |
| (7.4) Timer Queue Utilization Timing | 91 |
| (7.5) Results of the Prototype Hardware-in-the-Loop Simulation | 92 |
| (7.5.1) Simulation Response to Failures | 92 |

TABLE OF CONTENTS (CONTINUED)

| | Page |
|-------------------------------------------------|------|
| (7.5.2) Bo Value Errors | 94 |
| (7.5.3) Potential Resolution Problems | 94 |
| (8) Conclusions | 96 |
| Appendices | |
| (A1) Macro Routines Listing | 98 |
| (A2) RTMOS Software Listing | 105 |
| (A3) Applications Tasks Listing | 158 |
| (A4) Schematics | 180 |
| References | 184 |
| Vita | 187 |

LIST OF FIGURES

| Figure | | Page |
|--------|-------------------------------------------------------------------|------|
| 2.1 | URV TN21 | 6 |
| 4.1 | Interconnection of Multiprocessors | 14 |
| 4.2 | Degradation of Added Processor Power on a Single Bus | 16 |
| 4.3 | Parallel Busses | 18 |
| 4.4 | Interconnect of All I/O to All Processors . | 20 |
| 4.5 | URV Multiprocessor System Configuration . . | 22 |
| 4.6 | Distributed Shared Memory | 23 |
| 5.1 | Single Processor Queue Structure | 39 |
| 5.2 | Task State Transition Diagram | 47 |
| 5.3 | Layered Representation of the RTMOS | 49 |
| 6.1 | System Test Configuration | 62 |
| 6.2 | Ki Algorithm With Parallel and Sequential Parts | 67 |
| 6.3 | Data Flow Representation of Ki Algorithm . . | 68 |
| 6.4 | Remote Procedure Call Representation of Ki Algorithm | 69 |
| 6.5 | Two Processor Division of Tasks | 73 |
| 6.6 | Three Processor Division of Tasks | 74 |
| 6.7 | Layout of Wirewrap Board | 76 |
| 6.8 | Simulation Test Layout | 77 |
| 7.1 | Three Processor Division of Tasks With Timing Data | 89 |

PREFACE

Parallel processing research is still pretty much in its infancy. Much "hype" surrounds many of the efforts to date. Promises of high performance gains with unbelievable processor utilization efficiencies are not uncommon. Still, many potential applications exist, and many varying solutions to the problems are possible.

Serious research into the technological area still needs to be performed, particularly in the programming aspect of the problem. The best way to understand the technology, and the problems that exist in the application of the technology, is to get the "hands-on" experience in the development and use of parallel systems. I am grateful for the chance given to me by AFWAL/FIGL to gain this experience, both in this thesis effort and in my normal job duties.

Many varying disciplines are involved in a development effort such as this. I thank all those persons who provided advise or help in those areas where I needed it. Not all can be mentioned here, but certainly all are appreciated. Without them, this project could not have been a success. My sincerest gratitude to:

First and foremost, Dr. Kuldip Rattan, who acted not only as my thesis advisor, but also served as consultant for the control mixer applications functions.

Tom Roesle, who helped to construct the VME rack and power

supply system, and procure the necessary parts.

Doug Roy, who provided the necessary URV and 8061 autopilot information and helped set up the simulation runs.

Dr. R. D. Dixon and Dr. Alastair McAulay, who served on my thesis committee and provided valuable advice.

And last, but not least, the current generation "Microteers": Don Pogoda, Mike Rottman, Tom Dermis, Jeff Mangel, and Vince Crum, who provided draft reviews, hardware design advice, and idea critiques; and put up with me when things went the craziest.

(1) Introduction

The Unmanned Research Vehicle (URV) in-house program at the Air Force Flight Dynamics Laboratory (AFWAL/FI), Wright Patterson Air Force Base is developing a new research testbed to provide a wide range of capabilities in support of advanced, low cost flight testing of flight control concepts. Implicit to the development of the testbed vehicle, hereafter referred to as TN21, is the addition of an advanced on-board avionics system to support computationally intensive embedded tests. As it is with the rest of the vehicle, the avionics system is to be developed for high capability at low cost.

Single processor architectures, such as the one used in the current URV system, can be sufficient for basic autopilot control, but lack the necessary throughput potential for advanced embedded tests like those envisioned for future URV applications. The capabilities of microprocessors and microcontrollers are rapidly improving; however, the demands on digital systems are outracing this growth in improvement. For example, adaptive control algorithms and artificial intelligence (AI) are likely to drive throughput requirements an order of magnitude or more beyond previous generation requirements. In contrast, next generation processors can only be expected to be two to four times the performance of their predecessors [1,2,3]. As a result, multiple processor architectures will be required to meet the goals of advanced system needs and tests.

Multiprocessor systems are being utilized in or researched for many varied applications, including flight control [4,5,6]. The technology, though not yet mature, carries many possibilities, the most obvious being high speed computation. Many multiprocessor architectures have been proposed [7,8,9]; however, no one architecture has emerged as being superior to the others over a wide range of applications. At present, the application dictates the architecture used.

The purpose of this thesis project is to exploit advances in microprocessor and memory technology, bussing systems, and real-time multitasking operating systems techniques to develop a multiprocessor architecture appropriate for application to a URV system. This research will allow low cost flight testing of concepts which heretofore required high cost/high risk flight tests on expensive manned systems. To meet the goal, a first phase effort has been performed to develop and demonstrate a multiprocessor system suitable for use on the proposed URV. This system is a prototype, not the actual flight-worthy system. Not all aspects of the final URV avionics system have been designed and implemented. This work was focused on the main "computing engine" of the multiprocessor system and its associated operating system software. This focused, multiphase design strategy was taken to provide short term, low cost results with limited manpower. As such, development time, cost, and use of available laboratory resources were important drivers.

The prototype multiprocessor system has been developed and demonstrated incorporating near-state-of-the-art

microprocessors, coprocessors, and interconnect technologies. A real time multiprocessor/multitasking operating system (RTMOS) has been designed and implemented to coordinate the parallel tasks and data exchanges. To demonstrate the capabilities of the prototype, a set of applications tasks, implementing a control mixer algorithm for failed control surface reconfiguration, was developed.

To begin the description of the development of the research and prototype, Chapter 2 gives a background discussion of the related URV and Continuously Reconfiguring Multi-Microprocessor Flight Control System (CRMMFCS) programs. Chapter 3 covers the requirements driving the design of the multiprocessor system. Chapters 4 and 5 discuss the hardware and software design decisions respectively. Chapter 6 addresses the approach taken to develop and test the laboratory prototype. Chapter 7 concludes the technical discussion with prototype performance and demonstration results.

(2) Background

For several years, the Control Systems Development Branch (AFWAL/FIGL) of the Air Force Flight Dynamics Laboratory's Flight Control Division has been performing research utilizing Unmanned Research Vehicles (URV). Using an automotive emissions and fuel economy processor, the Intel/Ford 8061, the URV in-house program has developed a low cost digital autopilot which has provided significant size, weight, and power savings over its predecessor system. The extensive input/output (I/O) capabilities of the 8061 make it ideal as a "single chip" controller in such applications [10,11]. Included in these capabilities are thirteen analog-to-digital (A/D) conversion inputs, eight high speed inputs, and ten high speed outputs which can be used for pulse width modulated signals.

The URV has progressed to its current state as a low cost/risk in-house flight testbed for research projects at the Flight Dynamics Laboratory. However, due to the limited computational capabilities of the single 16 bit processor of the 8061, many new proposed tests must be run on a more powerful ground-based computer system, with control commands uplinked to the URV via its telemetry system. This process suffers from several limitations, the most critical being the relatively slow uplink transmission speed.

In its role as a low cost/risk flight testbed, the URV has been very successful. The capabilities of the URV system, however, must expand to meet its potential

applications if it is to continue to serve as a useful in-house research tool. The current airframe, originally designed as a drone with specific mission requirements, is limited in performance and adaptability. The airframe is relatively heavy, causing a high stall speed. The bulkhead is such that payload and electronics space is extremely limited. The use of the 8061 allows digital autopilot capabilities to be placed in the small space available, but no expandability exists to allow for growth into more advanced embedded tests.

To correct these shortcomings, a new URV testbed system (TN21) has been initiated (Figure 2.1). The design will incorporate a modular airframe structure that will allow different configurations, such as varying wing sizes, to be implemented around the baseline design. The aircraft will be made of lighter materials and will make better use of space to create less wing loading and greater maneuverability. The design will also create a significantly larger payload bay which can support more electronics and cargo. New control surfaces and an overall improved aerodynamic design will allow for a wider range of applications to be flight tested on the URV. In short, TN21 will be designed from the onset to be a flexible tool for low cost, high payoff flight tests.

In order to make the best use of the performance and flexibility capabilities of TN21, an avionics system with greater capabilities than the current autopilot is required. The increased space for embedded electronics has opened the possibility of an advanced, yet low cost, control system utilizing multiple microprocessors and expanded memory. The

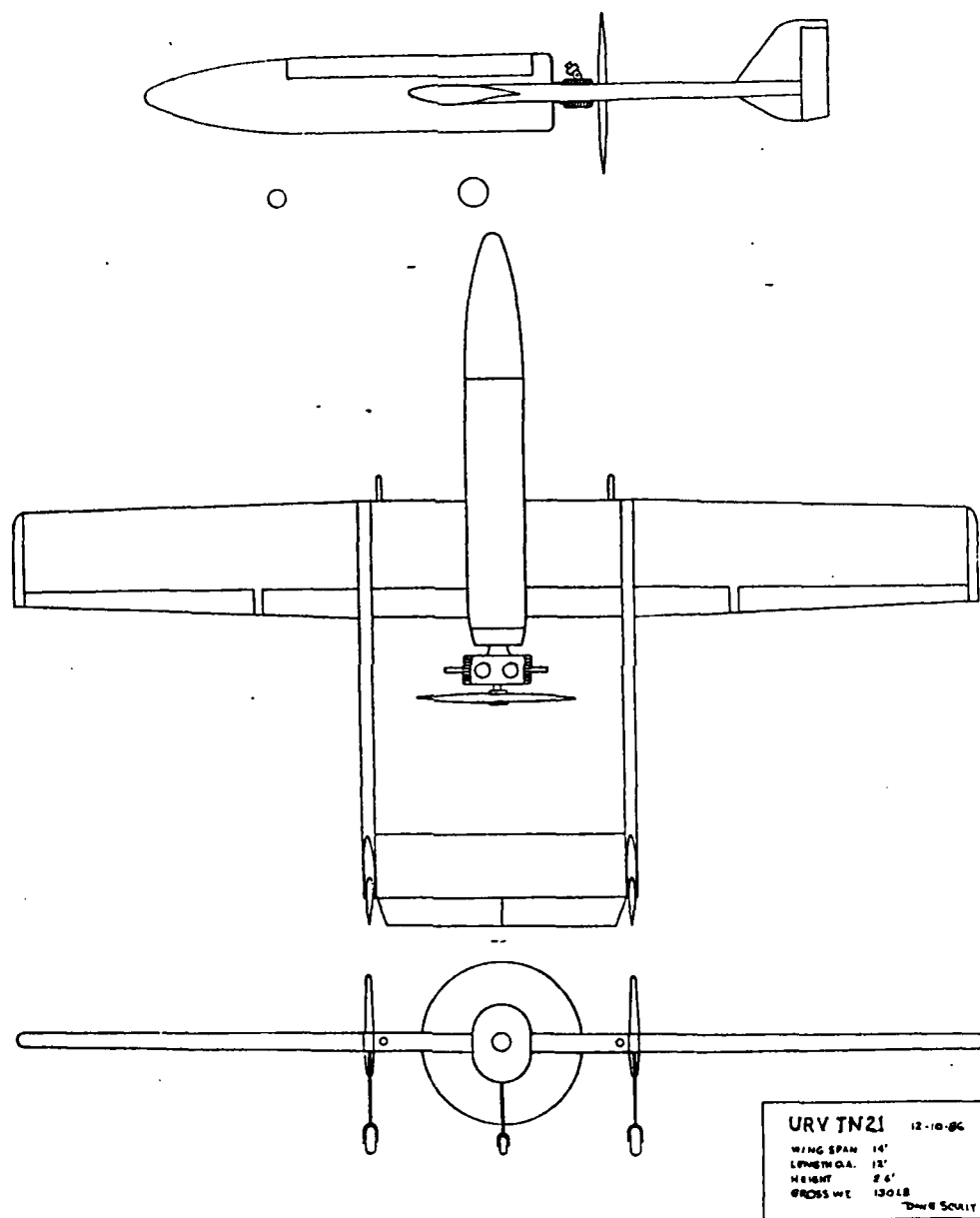


Figure 2.1

system should not only be able to handle the real time response needs of advanced control laws and the changing characteristics of a flexible and reconfigurable airframe, but must also be applicable to a wide variety of test problems. Already a wide spectrum of potential applications exists. Among these are control law reconfiguration around failed surfaces, fault tolerant hardware and software techniques, artificial intelligence, and the application of High Order Languages (HOL) such as ADA to real time control.

AFWAL/FIGL has accumulated the background knowledge and experience necessary for the development of such a system. Concurrent with the URV development work, AFWAL/FIGL has conducted research in the areas of microprocessor-based multiprocessor systems and parallel processing as applied to flight control and vehicle management systems. The Continuously Reconfiguring Multi-Microprocessor Flight Control System (CRMMFCS) in-house project (1980-83) produced several unique concepts in fault tolerance and parallel processing and a successful laboratory demonstration system [4]. This program has since spawned further research into microprocessor applications in flight control and related areas, including the current Advanced Multiprocessor Control Architecture Definition (AMCAD) in-house project [5]. These programs laid the groundwork for this thesis research by providing valuable hands-on experience in the design, development, troubleshooting, and programming of multiprocessor laboratory systems using microprocessor emulation systems, logic analyzers, and support software. Lessons learned during these other development efforts have

been applied to allow effective integration of the technology area to a low cost research testbed vehicle.

(3) Design Considerations of a Multiprocessor URV Avionics System

Because of the wide range of potential applications of such a URV system, a quantifiable performance measure of the avionics system is difficult to pinpoint as a baseline. Almost none of the tests identified in the previous chapter have been formalized into planned tests to this point. From this, flexibility appears to be the critical design goal. A multiple processor configuration is desired, but the number required is a function of individual processor capability, communications throughput, and applications computation requirements. Without the last, the best design is the one that allows for minimal multiprocessor configurations with growth potential to meet needs.

Size, weight, and power often drive the limits of growth potential. For the TN21 system, however, these aspects have minimal impact. The available electronics space has been approximated at 9 inches by 34 inches by 11 inches. Current microprocessor, memory, and interface logic densities allow considerable computing power to be included in the available space. The on-board power system will be able to supply more than 50 amps at 24 volts, and again, current device technology allows operation well below this constraint. A design constraint for weight is currently unavailable. However, all indications are that the weight of electronics and packaging filling the available space will be within the limits of the aerodynamic design.

The input and output (I/O) requirements perceived for TN21 do not increase significantly over those of existing URV. Although the figures will vary due to changing configurations, the baseline requirement is 8 analog sensors and 10 pulse width modulation driven servos. The 8061 processor used in the previous URV system contains internally the capabilities to handle the I/O requirements of the new system. As will be addressed further in the next chapter, the problem with using the 8061 as the processor type for the multiprocessor is its lack of general purpose applicability. Without floating point support, operating system aiding instructions, and a conventional memory interface, the 8061 lacks the capabilities to make it a flexible computation base in a multiprocessor environment.

Reliability and safety are concerns in a design such as this. In manned systems, fault tolerance is required to ensure safety of flight and to avoid the loss of expensive systems. The URV is a unique flight test bed in that the airframe and equipment are relatively inexpensive. The aircraft are flown in controlled areas where risk is minimized. A full suite of fault tolerance mechanisms, including the capability to maintain testing after computing resources fail, is an overly optimistic, if not self defeating, design goal. The inclusion of a complete set of fault tolerance mechanisms would only serve to drain available resources and drive the design and utilization costs to an unacceptable level. Still, the use of more complex, multiprocessor configurations creates a greater chance of individual component failure. As with the previous URV system, the answer lies in a fail safe mechanism which

allows the pilot of the vehicle to regain direct control of the vehicle in an unassisted mode of operation or degrade the aircraft control to a slow, circling return to the ground. As such, the design criteria allows for a means to bypass the multiprocessor system upon detection, by the system or pilot, of a failure in the control system.

In summary, the multiprocessor system should be flexible to meet changing airframe and test configurations, utilize processors of general purpose applicability, and implement only a minimal set of fault tolerance capabilities. Most importantly, however, the system must provide efficient, high speed computing at a low overall system development and maintenance cost. By meeting these requirements, the multiprocessor avionics system will support the goals of the new URV research testbed.

(4) Overall Goals to Architecture Specification

The three basic areas comprising the avionics system can be categorized as computation, interprocessor communications, and I/O. This statement is not to imply that the three areas need be distinct; in fact, in some implementations quite a bit of overlap exists. In this design, however, the areas are best handled separately. The following sections describe the specification of the three areas and the resulting TN21 multiprocessor configuration.

(4.1) Computation Area

The use of the URV as a general test-bed for low cost/risk flight testing dictates the use of a homogeneous set of processors of a type that can handle a wide variety of jobs. The baseline requirement includes capabilities to handle real time operating system functions, 16 or 32 bit integer processing, floating point operations, logical bit manipulations, and a variety of memory addressing modes. These are not difficult criteria to meet; in fact several current, readily accessible microprocessors exist which contain the above capabilities. Examples include the Motorola 68000 family, the Intel 8086/286/386 family, the National 32x32 family, and the Zilog Z8000/Z80000 family. To complicate matters, none of the acceptable microprocessor families has a clear technological advantage over the others as applied to the wide range of applications. The 8061, however, does not meet the stated goals. The chip was

designed for microcontroller applications, not for general purpose computing.

As a result, the processor of choice was picked due to the development support available in the Microprocessor Laboratory at AFWAL/FIGL. This facility utilizes MC68000 in-circuit emulators and logic analysis capabilities in a variety of programs and has assemblers for the development of MC68000 code. The use of these capabilities eases the development cycle and results in a more reliable designs which take less time to develop. The choice of the MC68000, therefore, provides the required capabilities while allowing for low cost development.

(4.2) Interprocessor Communications Area

Many schemes have been developed or proposed for the interconnection of multiprocessors [7,8,9]. These range from a simple, single bus to a complex, multiple interconnection network. Figure 4.1 shows some of the possibilities. Intuitively, the development of a system for a URV would not include a complex, difficult to develop and test, interconnection scheme. In fact, the wide availability of commercial busses and parts leads to the choice of a bus. Two questions must first be addressed before finalizing this choice.

The reliability of a single thread bus can be seen as a potential problem. However, as addressed in the previous chapter, multiple redundant channels of communication, providing a fault tolerant capability, are not required in this URV system. Although a commercial bus with multiple processors attached is more likely to fail than the single

| Interconnect Scheme | Subtype | Specifics | Complexity | Comments |
|---------------------|----------------------|------------------------------------------------|--------------|---------------------------------------------------------------------------|
| Busses | Parallel | Arbitration req'd ex: bus request/ grant | Low | Commercially available in complete packages |
| | Serial | Arbitration req'd ex: token passing | Med | Some parts available |
| Links | Ring | Active bus components | Med | Complicated bus transceivers, some parts available |
| | Fully Connected | Serial (n-1) per proc | Med- High | Not practical with more than a few processors |
| | Hypercube | Serial $\log(n)$ per proc | Med- High | Requires transceivers with forwarding capability |
| Networks | Butterfly Shuffle | Multiple levels of switches | High | For low arbitration situations, otherwise complexity is too high |

Figure 4.1

processor URV system of before, a fail-safe mechanism built around the multiprocessor "network" can provide the reliability required.

A more troubling question involves a common concern in single bus multiprocessor systems: transfer bottlenecks. It is commonly accepted that a practical limit exists on the number of processors attached to a single bus. Beyond this limit, the addition of more processors degrades, rather than improves, system performance by forcing more resource sharing. Even before this limit is reached, degradation of added processor power can be significant. Figure 4.2 shows a realistic graph of processor numbers versus effective processing power. Note that the ideal limit is a linear increase in processing power with increased numbers of processors. This situation is not reached in practice for a variety of reasons; the overhead of achieving parallel interactions being a major cause. The result is a reduction in the added percentage of processor power as more processors are included in the system. However, if the number of processors attached to a single bus is kept low enough and the speed of the bus is sufficient, a single bus can be used to support multiple processors. Ideal parallelism may not be achievable, but effective gains in computing power can be realized.

Given the sizing limits from the previous chapter, six to eight processors appear to be the practical upper end for the number of processors to be used. This limit would appear to be sufficient for all foreseeable URV embedded applications. Keeping the granularity of computation to a

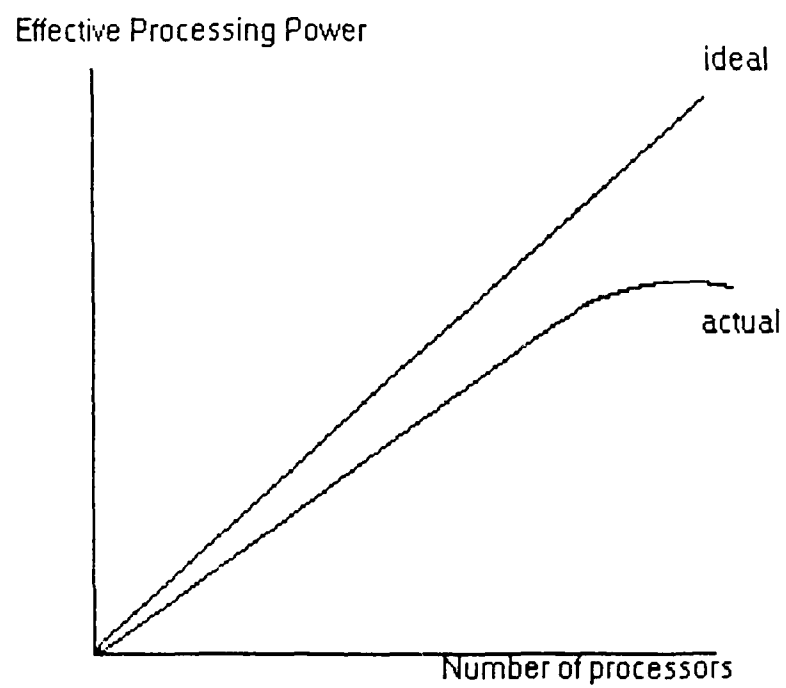


Figure 4.2

coarse grained level with transfers kept to a minimum, six to eight processors can be used practically in the URV system with a single bus interconnect.

As with the choice of processors, there are multiple, readily available options for parallel busses (Figure 4.3). None of these options is unworkable with the MC68000. However, the VME bus is the most compatible, being developed by Motorola with a protocol almost identical to the MC68000. Also, far more available components based on the MC68000 exist for the VME bus than there are for any other parallel bus. The VME bus meets the requirement for high speed transfer with a near state-of-the-art 40 Mbytes per second peak transfer rate [12]. The bus specification also includes features of access fairness and interprocessor interrupts. The asynchronous transfer protocol allows the addition of components which operate at different speeds than the bus itself. This feature can be useful in an environment where flexibility is desirable. Also, industry acceptance of the bus has led to wide availability of parts, boards, and assemblies. Of the other bus options available, only Multibus II appears to have the wide industry acceptance, capability, and availability of components of the VME bus. Comparisons of the two busses in industry publications [12,13,14] have produced no clear advantages for either. As such, the choice of processor led to the choice of the VME bus for the URV multiprocessor prototype.

(4.3) I/O Area

The introduction to this chapter described the I/O

| Parallel Bus | Address Width | Data Width | Type | Multiplexed | Multiprocessor Capability |
|----------------------|---------------|------------|-------|-------------|---------------------------|
| VME | 16/24/32 | 8/16/32 | Async | No | Yes |
| Multibus II up to 32 | 8/16/24/32 | 8/16/24/32 | Sync | Yes | Yes |
| Multibus I | 8/16/20/32 | 8/16 | Async | No | Yes |
| NuBus | 32 | 32 | Sync | Yes | Yes |
| S-100 | 16/24 | 8/16 | Async | No | Yes |
| IBM-PC | 20 | 8 | Async | No | No |

Figure 4.3

section as a separate, distinct unit. I/O capabilities could be merged into the processing modules of the multiprocessor, but this is not practical from a number of standpoints. First, connection of all I/O devices (sensors, actuators) to all processors (Figure 4.4) becomes cumbersome, and expandability is quickly inhibited. Dedicating certain I/O devices to certain processors limits the flexibility of processor utilization. This configuration could also hinder expandability and the programmability of the system. Also, the inclusion of I/O capability, such as analog-to-digital converters, would severely complicate the processing module hardware; creating more hardware for no corresponding gain in processing capability. Usage of the 8061 could solve this problem; however, as stated previously, the 8061 is basically unsuitable as the computation area processor-type for the multiprocessor system.

A more acceptable solution is to separate the I/O area from the main computational area. Although the 8061 is not suitable for a multiprocessor environment, interface to the VME bus is still possible. The solution, therefore, is to connect all I/O to an 8061 module (or multiple modules if needed) and program the 8061 to supply the multiprocessor system with digital inputs through the VME bus. Outputs from the system can then be sent back along the bus to the 8061 for pulse width modulated output to the servo actuators.

One byproduct of this choice is that it provides a possible solution to the fail safe operation considerations discussed above. If provided with a "bare-bones" autopilot function in reserve, the 8061 can be commanded by the pilot

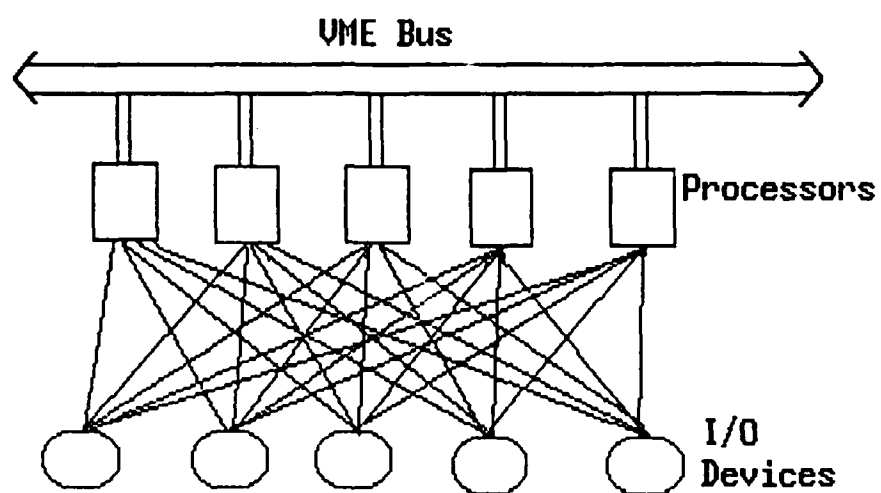


Figure 4.4

to take control of the aircraft in the event of a failure in the multiprocessor section. In this scenario, all tests would be terminated and the vehicle would be returned to the ground safely. This mode provides a reliability no worse than that of the current autopilot.

(4.4) Resulting Configuration and Hardware Selection

The resulting hardware configuration is shown in Figure 4.5. As noted in Chapter 1, the main design drivers for the prototype were development time and cost. These drivers dictate the use of available parts where possible. The decision made was to purchase as much of the hardware preassembled as possible to limit debugging time. MC68000 processor boards with VME interface were purchased [15]. These boards contain 512 Kbytes of zero wait state dual ported dynamic RAM accessible from the VME interface as well as the MC68000 resident on the board. The boards also include 128 Kbytes of EPROM and an interface connector through which a wire-wrap board can be attached for hardware expansion. A VME backplane, rack assembly, and power supply were also purchased.

The dual ported RAM sections on each board provide the means for interprocessor communication. Since each section can be set up to be addressed in a different memory range on the VME bus, the concatenation of the sections creates a distributed version of a shared memory. Figure 4.6 shows the logical view of the VME addressable memory area.

The selection of boards utilizing this communications technique was not arbitrary. Experience with multiprocessor

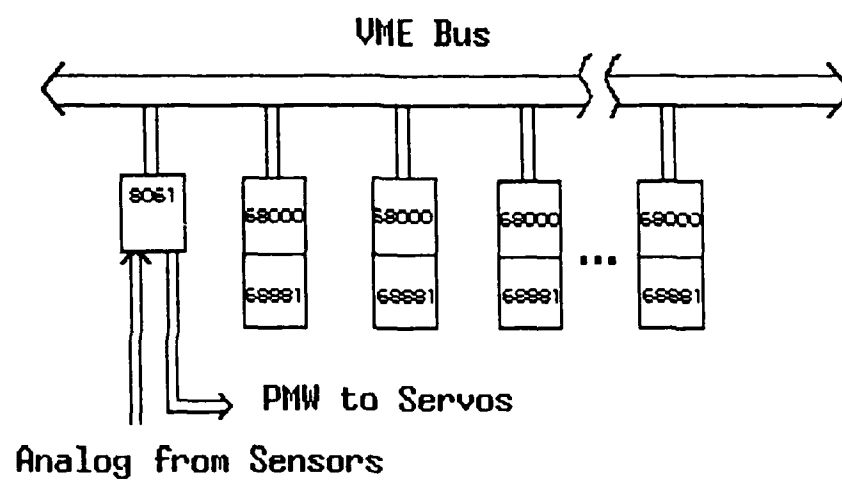


Figure 4.5

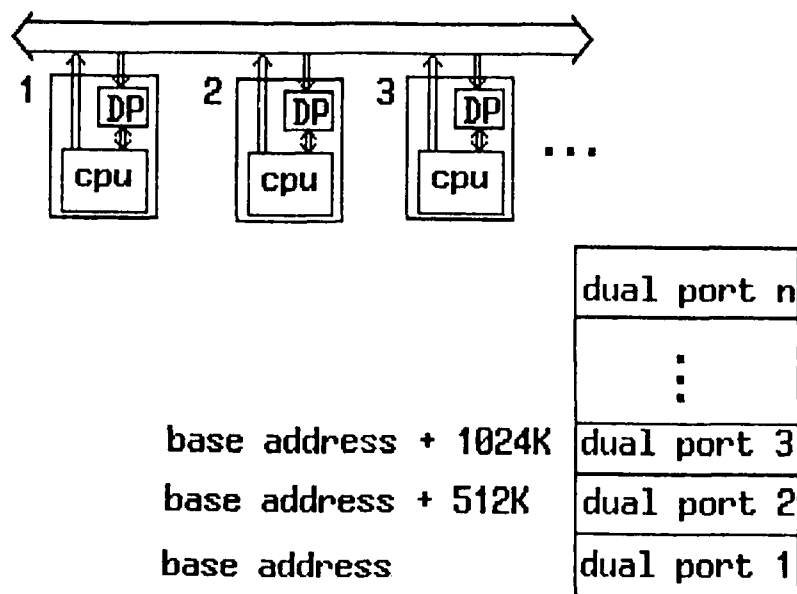


Figure 4.6

system software at AFWAL/FIGL has indicated that shared memory techniques represent a simpler view to the programmer and provide the same with more flexibility than message passing (or point to point) techniques. One example of the flexibility possible is the capability for an any-task-on-any-processor programming model. This capability allows the programmer to design the parallel software without specific details of the architecture, such as the number of processor modules. Chapter 5 will address this shared memory capability. If required, however, a message passing logical view can be set up over a physical shared memory design through the use of mailboxes or other similar data structures.

(5) Software Specification

The software for the TN21 avionics system intersects with two often distinct areas of software development: real time control and multiple processor operating systems. The commercial market contains skeletal real time operating system kernels which are usually designed to operate on a single processor. Development of multiprocessor operating systems are often taken with more concern towards keeping processing elements busy with some portion of the task loading and coordinating exchanges of messages than towards the rapid response to asynchronous events or the strict periodic sample/compute/output cycle of real time control. The software development of the URV multiprocessor system has been focused towards applying real time operating system techniques in the environment of a multiprocessor system.

(5.1) Techniques of Multiprocessor Software

Multiprocessor software can be viewed in many ways. In the simplest case, each processor can be given a distinct job to perform, with little or no interaction with the other processors of the system. Each processor's job is a separately specified and developed piece of software code. An example of this is a multiprocessor system supporting batch processing of user programs. Each program submitted can be allocated to a separate processor. This method is dependant upon separate, independant threads of computation, and often results in inefficient use of processing

resources. Some processors are idle, waiting for work, while others are busy performing jobs which conceivably could be subdivided.

From this simple case, many extensions are possible. One example is the static assignment of multiple, sequentially ordered jobs per processor. Instead of having each processor perform a single, complete job, each now may participate in multiple threads of computation by executing subsets of these threads. This brings the interactions closer to true parallel processing and allows for more even processing load distribution. The interactions of the processors are still preplanned according to the job scheduling. Execution of program sections is totally determined before run-time, forcing the programmer to be the scheduler. Code execution time is another consideration the programmer has to be concerned with, so that exchange windows can be met. Although efficiency of processor utilization is made better, coding is much more difficult.

Single processor multitasking operating systems can provide the basis for another option. These have been used extensively for real time control and can provide for the efficient and timely scheduling of jobs. The question is how to extend this type of model to multiple processor systems.

Two methods used are either to utilize a single queue of tasks or a single task resource manager from which the jobs can be obtained. The latter case can be considered a form of master/slave processing, where one processor of the system is dedicated as a distributor of tasks to the other processors. The former case requires a shared memory with mechanisms for insuring uninterrupted access by processors

to the queue during task acquisition. Neither of these cases really follows directly, in implementation, from the single processor multitasking model.

If each processor is given its own queue of tasks to perform, the single processor model can be used. The question to address is how tasks are distributed to the individual queues. The assignment may be static and determined during programming, or dynamic and changeable during execution. Also, given that the tasks may not be prescheduled for exchanges, how are interactions coordinated?

(5.2) Programmer Viewpoint: Tasks

A task is a unit of software which may operate in parallel with or sequentially in coordination with other tasks. The code may be thought of in the same general terms as a module or subroutine in conventional programming. A multitasking operating system provides for the scheduling of multiple tasks for the effective operation of the system.

Tasks are not often independent. They interact with other tasks to acquire data, and to enforce cause and effect relationships. The division of a problem into a set of tasks, therefore, is similiar to the division of a conventional program into a hierarchy of subroutines. Software is organized into modules of logical relevance, and their interfaces to, or interactions with, other modules are specified. The difference with tasks is the additional factor of time. Certain tasks can operate at the same time, while others operate in some ordering defined by task

interactions. A mechanism to enforce data exchanges according to the interaction specification is required.

The above discussion addresses the division of a problem into tasks with interactions defined. The mapping of tasks to processors was intentionally left out. If the system is comprised of a set of homogeneous processing modules, and if a shared memory architecture is used, a task can be made to run on any processor. Interactions are made to the shared memory, not to specifically addressed processors. With these assumptions, the programmer can design the tasks and interactions without any knowledge of the number of processors or specific assignment of tasks to processors. This any-task-on-any-processor scenario eases the software development for the multiprocessor by allowing the programmer to concentrate on the tasks and interactions required, not the architecture used.

(5.3) Analysis of Available Real Time Operating Systems (RTOS)

Given the desire to use off-the-shelf components where appropriate, an analysis of existing RTOS's [16] was required. Four packages developed for use in MC68000 systems were chosen: VRTX from Ready Systems [17], MTOS68K from Industrial Programming Inc. [18], RMS68K from Motorola [19], and PSOS/PRISM from Software Components Group [20,21]. The most interesting result of the study was that the user interfaces in these packages vary little from one to another. The main differences in the interfaces occur in the communication and synchronization primitives. These primitives will be discussed first, then discussion of the

common functionalities will follow.

Three basic data structures are used. The semaphore is probably the most studied and written about construct for mutual exclusion and intertask synchronization. In its simplest form, a semaphore is a "flag" on which two operations can be performed. The "P" operation will allow processing to proceed if the flag is in the "pass" state, but "stops" processing otherwise. The semaphore can be used to protect a resource, or insure mutual exclusion. One task sets the flag and proceeds to use the resource. Another task which attempts the "P" operation after this point will encounter the "stop" state, and as such, is prevented from proceeding in using the resource. When the task is done with the resource, the second operation, "V", is used to set the flag back to the "pass" state so that a waiting task can use the resource. Other variants of semaphores exist, including counting semaphores which are used to manage multiple resources. [22,23] provide more complete descriptions of semaphores and their variants.

Event flags are groups of simple flags. Each flag signals whether a particular event has or has not occurred. Processes can wait on event flags much like for semaphores, although operations such as "P" and "V" above are not defined specifically for them. What characterizes event flags is the ability to combine waiting on multiple event flags simultaneously with "AND" and "OR" operations.

Mailboxes are a higher level construct. Although providing a wait-on capability like for the structures above, mailboxes also provide an inherent means to pass

data. A "post" operation to a mailbox is similar to the semaphore's "V" operation, except that a pointer (or unit of data) is stored as a part of the operation. A "pend" operation corresponds to the "P" operation, with the addition of receiving the previously stored pointer (or data).

PSOS and MTOS utilize all of the above constructs; and, as such, provide the most flexibility to the user. VRTX only provides support for mailboxes. The argument given for this is that the functionality of event flags and semaphores can be emulated with mailboxes [24]. The use of a single structure provides a common interface to the programmer. RMS68K only supports semaphores.

Most of the user interface functions are common to all of the above packages; the specific operations on the data structures above being the primary exception. Typical functions include task creation, deletion, suspension, and resumption; memory management; get and set system time; and modify task priority. All use basically the same task state model and utilize a real time clock or counter which is accessible by the user.

The most important characteristic for this study, however, is the support for multiple processors. RMS68K, as described in the literature, does not address this capability. VRTX does not support multiprocessors directly; however, Ready Systems does describe a way to utilize their single processor product on multiple processors with shared memory and interprocessor interrupts [25]. MTOS supports multiprocessors, but the method is for systems utilizing shared memory in a "tightly coupled" fashion. The multiple

processors acquire their tasks from a single (central) queue. Software Components Group uses a modular, building block approach to their package. PSOS is the multitasking kernel located on each processor in the system. An additional package, PRISM, is added to provide multiprocessor capability. Like MTOS, PRISM requires a bus based architecture and shared memory. Interprocessor interrupts are desired, however, polling can be used if they are unavailable. Tasks communicate by "datagrams" which are queued up in each processor. Tasks need to have knowledge of the destination task's processor identification number. This requirement is actually transparent to the application programmer through the use of a "global name server" which converts a logical "name" to physical address, however, it appears that the "server" must be updated when tasks are assigned to processors.

(5.4) Analysis of the Multiprocessor/Multitasking Problem in the Control Environment

(5.4.1) The Multiple Processor Problem

Of the above, PSOS/PRISM and VRTX appear to correspond the best to the model required. Interestingly, both sets of literature indicate that two techniques are possible to accomplish the multiple processor communication and synchronization: interprocessor interrupt and polling [21,25]. To explain the need for either of these techniques, consider the following scenario. A task on processor n is waiting for data from another task. Let us assume that this

task is resident on another processor *m*. If each processor has its own set of task queues and the task on processor *n* waits for the data (either by going to a wait queue or polling), how does the task on processor *m* signal to the other task that the data is ready? This situation is complicated further if the task on *n* is put to "sleep" waiting for the data, allowing another task to use the processor. In the single processor multitasking model, the process is accomplished via an operation, such as "V" above, that will free the waiting task from the wait queue once the data is available. Across multiple processors, however, one processor should not have access (modification rights) to another's task queues. Some sort of signal, such as an interrupt, is required for one local kernel to indicate to another that this operation needs to be performed and which task is to be signalled.

Interprocessor interrupts provide the most direct way to accomplish this. An interrupt service routine on each processor can be set up to act much like a local "V" operation. Limitations exist, however. The interrupt scheme in hardware must be any-to-any in order to support the any-task-on-any-processor scenerio. Also, knowledge of a "consumer" task's processor must be available in order to activate the proper interrupt. Interrupting all processors in a broadcast manner is wasteful and inefficient. The knowledge that is required does not preclude the desire for any-task-on-any-processor, but does create a need for a method to tag tasks with a processor identification number when they are activated on that processor, and to make this available as part of the communications/waiting process.

Polling, as the alternative to interrupts, is often discarded quickly as being inefficient. A task that is waiting for data, and stays on the processor or ready task queue while waiting for a data available indication, would appear to waste precious processor time busy waiting. Also, the context switch time in repeatedly bring a polling task onto a processor, then removing it when the data is unavailable, can be significant. In short, the disadvantages involve multiple passes through the processor while polling, each with a costly context switch time. The advantages include a much simpler means of implementing the any-task-on-any-processor scenerio, without any regard for processor identification.

The two methods above assume a coordinated transfer of data between tasks. Prescheduled tasks with timed transfers, as discussed earilier in this chapter, is an example of an exchange technique without coordination, or handshaking. This technique requires much programmer precompilation knowledge of execution time and task/processor placement.

(5.4.2) Intertask Communications

Several levels of coordinated transfers can be used. The simplest case involves a basic flag which signals when data is ready. This unilateral rendezvous [24] has only the consumer of the data wait in the exchange. Producers can update at any time and do not wait for the consumer to respond. Bilateral rendezvous involve both producers and consumers waiting until both are ready. The bilateral transfer provides the most reliable exchange since it is

assumed that both sides of the transfer are active and meet during the same time window. Unilateral transfers require much less overhead to effect the exchange.

(5.4.3) Control Timing

Digital real time control systems usually are characterized by two types of processing: interrupt driven, asynchronous response to external events and/or periodic sample/compute/output sequences. The latter case is of primary concern in the case of the URV system. The resultant requirement is for a means to schedule tasks on a strict periodic basis. Two methods have been identified to handle this situation. The assumptions made from the previous discussions are that tasks are coordinated in a rendezvous-like fashion and that a separate I/O section exists to provide the tasks with input data and to take task outputs to convert to servo actuator signals.

In the first case, tasks can be ordered in a dataflow-like manner in which the output of one task is required before another can start. The processing of a task, therefore, is characterized by one or more sequences of: receive inputs, compute, send outputs. The I/O section can be programmed to handle interrupts or loop in a timed manner to sample data. The I/O section then makes a rendezvous with the first computational section task (or set of tasks) in the control law computations. This task then "fires" other tasks, and so on. When the tasks complete, they go into a wait loop, awaiting the next rendezvous to start again.

An obvious problem with this case is that tasks are spending precious processor time waiting for a rendezvous (a

polling type of situation) in order to start the next computation. A better method is to utilize a special wait queue to hold tasks for a specified period of time. This method requires a timer or counter function to be available in the kernel. The kernel monitors the queue and the timer to determine when tasks are to be removed. Task periodicity is then accomplished by placing the tasks on the queue once their computation is complete. Once there, they are suspended for a specified period of time (the period of control computation) until the kernel removes them. The dataflow-type rendezvous method of the first case can then be used to insure proper task sequencing.

(5.5) URV Real Time Multiprocessor/multitasking Operating System (RTMOS) Kernel Specification

Although PSOS and VRTX may have been suitable for use in the URV multiprocessor system, several factors led to the decision to design and develop a multitasking kernel instead of purchasing one. Cost was one such factor. [16] gives typical price information. Note that source code, far more costly than object code, is required if modifications are needed. The most critical factors were the learning curve and porting-to-target times. Unlike an off-the-shelf processor board, a commercial RTOS is not usable when it arrives. The software must be ported to the target processor system by providing the software "hooks" between the provided code and target resources, such as RAM and timers. The code may also require transfer to new ROM chips to accommodate the target board design. The time taken to port

the new code and learn how to interface to it is significant; much more than setting up an off-the-shelf processor board. Because of these factors, a short study investigated those features needed in the URV kernel. The results of the study were that the functions required were fewer in number than commercial RTOS's and not difficult to implement, and that an in-house developed kernel would provide the flexibility and ease of use required. Upon development of the kernel features, as discussed below, a further discovery was made that performance improvements could be realized by tailoring the kernel to the application and target hardware. The output of the resultant development effort was the URV Real-Time Multiprocessor/multitasking Operating System (RTMOS), a kernel which operates identically on each processor in the system with the added capabilities of multiprocessor interactions and synchronization.

(5.5.1) Kernel Structures

This section describes the resultant kernel and intertask communications specification for the URV RTMOS. To begin, let us review the underlying hardware structure. The computational section is comprised of a set of MC68000 processor boards interconnected by a VME backplane bus. The basic means of interprocessor communication is shared memory comprised of the uniquely addressable dual port memory segments on each processor board. An interprocessor interrupt capability is specified for VME and can be implemented as any-to-any. However, the interrupt scheme requires knowledge (processor number or address) of the

destination of the signal. On each board is a resident MC68681 DUART chip to be used for serial communication between the processor board and an external terminal. This chip also contains internal timers.

As discussed earlier, the single processor multitasking kernel model is used as a baseline. This means that each processor has its own set of task queues. The queues are loaded with some subset of the total set of tasks. Any task can operate on any processor without modification or precompilation knowledge of its own or any other task's processor. This allows the set of tasks to operate on one or any number of processors by simply loading the tasks queues of each processor available with some subset.

In the single processor multitasking model, a task ready queue is used to hold and order tasks awaiting processor execution time. Priorities are generally associated with tasks so that ordering of the queue will reflect the relative importance or any time constraints of the tasks needing processor time. The URV RTMOS kernel builds upon this basic structure.

In addition to the ready queue, a timer queue is used. This queue, described in the previous section, is used to hold tasks for a specified period of time. The queue is ordered by release time rather than by priority. This ordering allows easy scanning for task release time since the search proceeds only from the front entry to the first which is not ready to be released.

The use of a timer queue requires a timer or clock function to be implemented within the kernel. The DUART chip

mentioned above allows for this to be added. The chip can be programmed to interrupt the processor at periodic intervals. In the URV RTMOS kernel, this interval was chosen to be one millisecond so that control loop or task frequencies of up to 1000 Hz can be implemented. The interrupt routine, referred to as the tick function, updates a location in memory as a counter. This location is used as a clock relative to system start-up. All task timings are based upon this clock.

A diagram of the queue structure of a single processor is given in Figure 5.1.

(5.5.2) Task Data Exchanges and Context Switches

Task data exchanges are handled via a variant of a unilateral rendezvous. The data structure used in the exchange is based upon the mailbox concept described above. The structure is comprised of four parts. The first part is a simple semaphore which is used to ensure mutual exclusion over the rest of the construct. The second part is a flag which enforces the producer and consumer relationship of the rendezvous. The third variable is a count of data words to follow in the structure. As such, the data being passed in the exchange comprises the fourth part.

Like the unilateral rendezvous described previously, the URV RTMOS utilizes exchanges where only consumers of data wait. Before waiting, a timeout clock value is recorded so that the wait time is bounded. The difference from the basic unilateral transfer comes in the use of the second piece of the data structure, the producer/consumer flag.

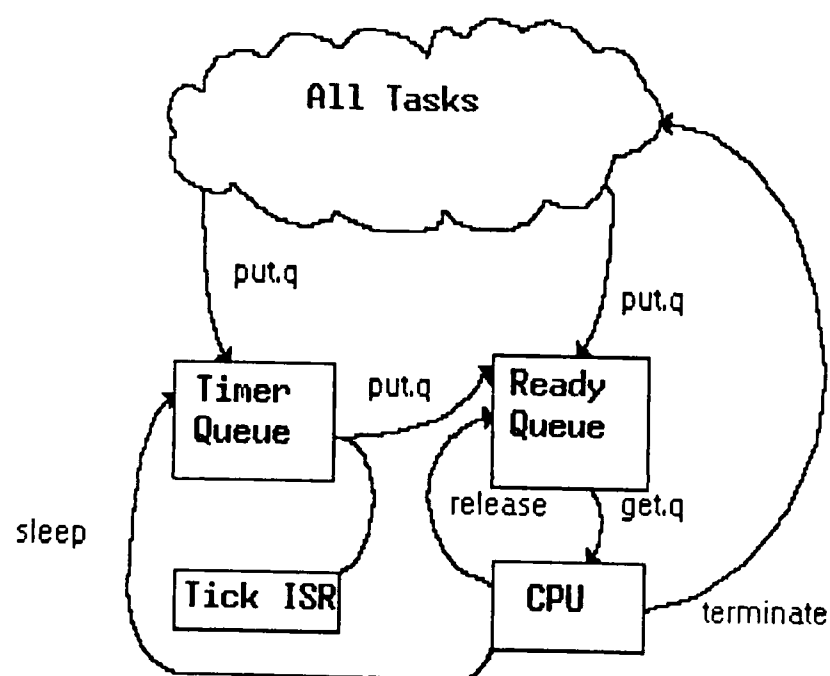


Figure 5.1

This flag has three states: producer's turn, consumer's turn, and failed exchange. The first two states are self-explanatory. The last is used in the event of a timeout or some other failure event. If the consumer detects a timeout, it can signal to the producer, through the flag, that data was not received within the prescribed time window. In this way, late producers or late consumers are detected and transfers cease until corrective measures are taken to "resynchronize" the tasks. This process will be addressed later.

As mentioned previously, polling is often considered to be an undesirable option in multitasking systems. The reasons for this include costly context switching times and "clogging" of the ready queue. Generally, a context switch involves a switch to the executive (via a jump, subroutine call, or software interrupt), a saving of the state of the current running task (its context), the removal and storage elsewhere of this task, and the acquisition of the next ready task. This new task's state is then restored and control transferred back to the user task mode of operation. In terms of the MC68000, which of these components are the most costly? Queue management, for simple queues such as described above, does not require extensive processor time. Mode changing and state saving/retrieving can, however. The commercial packages described earlier change from user to executive mode with a TRAP (or software interrupt) instruction. The cost is in the partial storage of the processor state taken by the MC68000 upon interrupt; and this process is performed each time a system function is

called.

The task state storage/retrieval is the main concern in context switches. The MC68000 has sixteen 32-bit internal registers. Since the kernel has no means of determining which registers the exiting task is using, it must save all of them. This makes polling undesirable since one failed polling attempt results in 32 words of data (minimum) to be stored, then later retrieved.

Two flaws exist in the techniques described above. First, a TRAP should not be performed each time a data exchange is to be made. The purpose for the TRAP is to force the processing into the kernel for proper and hidden access to RTMOS resources such as the task queues. Having the kernel also preside over data exchanges forces unnecessary overhead. Other options exist to hide data structure details from the applications programmer. These will be discussed later.

The other flaw is the storage/retrieval of all registers each time a polling iteration occurs. The context need only be stored and restored once. The task's registers can be saved at the first failed poll and be retrieved only after the poll is successful. In between, the registers are not used. Also, since the kernel is unable to determine on its own which registers are being used, a less brute force solution is to have the task specify the registers in use; or better yet, make the task save its own registers. This option is possible if a task's stack pointer points to within its own Process Control Block (PCB), a data structure used by the kernel for task information storage and queue linkage. Again, this process can be made transparent to the

applications programmer without putting the burden, and the overhead, into the kernel.

As can be seen from the discussion above, the functionality of the kernel has been limited basically to the management of the multitasking queues (ready,timer) and the handling of interrupts. How do we keep the burden of underlying communications data structures and polling from the applications programmer without placing it within the kernel? In the case of the URV RTMOS, where assembly language has been used in the initial stages of code development, macros are one possibility. Macros provide a generic unit of code with "gaps" in which specific instantiation information can be placed. In place of a section of code which implements a polling sequence on a particular data structure, a macro instruction can be used. Parameters in the polling macro instruction specify the data structure name (label), the registers in use, and the source or destination of exchange data. Macro instruction libraries can be supplied to applications programmers as easily as system functions implemented as TRAP's. In effect, macros are similar to additional, higher level instructions provided for use by the applications programmer. In higher order language implementations, special procedure calls can be utilized for the same purpose.

To demonstrate, the macro instructions for the producer and consumer polling will be discussed. First in algorithmic form:

Producer

```

P[var(sem)]
If var(P/C) <> C then
    Report Error to System
    var(P/C)=C
Else
    Put var(data)
    var(P/C)=P
End if
V[var(sem)]

```

Consumer

```

P[var(sem)]
Save registers
While (var(P/C)=C) and
(not timeout) do
    V[var(sem)]
    Wait
    P[var(sem)]
End while
If var(P/C) = P then
    var(P/C) = C
Else
    var(P/C) = F
    Report Error to System
End if
Get var(data)
Retrieve registers
V[var(sem)]

```

Note that the algorithms handle the mutual exclusion over the data structure, the storage/retrieval of specified registers, the checking/updating of the producer/consumer flag, the monitoring of exchange timeouts, and the transfer of data to and from the exchange data structure. Only the "Wait" operation is an actual call to the kernel. The rest of the algorithm is actually performed in user (applications) mode, but is hidden in source code from the applications programmer by the use of macro instructions. This programmer would implement a consumer poll with the macro instruction c.poll such as in the following example:

```

c.poll    tasklink,d0-d3/a5,mydata
where tasklink = exchange data structure
      d0-d3/a5 = specifies that d0,d1,d2,d3,a5 are
                  to be saved
      mydata   = destination of data from exchange

```

The source code to c.poll and the corresponding p.poll macro are included in Appendix (A1).

The question remains on whether polling is a viable option. Context switch times have been reduced (as will be discussed in a later chapter). Kernel calls have been reduced in number and have been made simpler. The implementation of any-task-on-any-processor is also simplified. Still, the potential problem of polling numerous times before succeeding is troublesome. The answer may lie in the proper use of the timer queue.

The situation to avoid when polling is to have a consumer of data begin waiting much before the producer is able to have it available. In other words, the desire is for a means to appropriately "order" producers and consumers. The extreme of having the applications programmer time schedule tasks in advance has been previously discarded. However, the ability exists to stagger the release times of tasks from the timer queue such that tasks at the the "front" of a thread of computation are released before those at the "end". This staggering need not be precise; in fact, the indeterminacy of the queues' orderings makes this extremely difficult, if not impossible. However, the desire is only to reduce polling to one or a couple of tries. With the timer queue described above, this situation does not appear difficult to realize. Experience in actual usage is required to demonstrate the practicality of polling with timer queues.

Another feature has been built into the RTMOS to limit the burden on the applications programmer. To prevent the programmer from having to design in points to release the processor to other tasks in order to create fairness, processor time slicing was added. This feature will

automatically switch out tasks which hold a processor for an extended period of time. In this way, long running tasks, such as those with multiple nested loops, can be run without "starving" other tasks on the processor. Not all tasks or sections of code within tasks should be time sliced, however. An example is the polling macros above. If sliced during one of these macros, a task will hold "ownership" of the semaphore, and likewise the variable, while the task is reawaiting processor time in the ready queue. To prevent this and other similar situations, each task PCB contains a time slice inhibit flag. System functions which turn on and turn off the time slicing privilege are used around sections of code where a slice can cause problems to occur. For example, these functions have been incorporated into the macro instructions previously described. The flag can also be put in the inhibit state at initialization and left untouched. This feature permits an "unsliceable" task.

The resulting user interface to the kernel is comprised of seven system calls: release (or exit), release-on-poll, sleep, terminate, report-to-system, slice-on, and slice-off. Release and release-on-poll cause the current running task to be removed from the processor and placed at the end of the ready queue. The next ready task is then assigned the processor. In release-on-poll, additional functions are performed particular to a polling task. Sleep does the same as release except that the previously running task is placed into the timer queue rather than the ready queue.

Terminate does not place the previous task on any queue; as a result, the task effectively "dies" rather than

is suspended, where reactivation is assumed. This function is used for nonrecurring tasks which are started up at initialization or by the RTMOS at certain times for single run operation. One case where this may occur is when a task makes a report-to-system system call. The report-to-system call is used to report error conditions, such as timeouts, to the RTMOS so that corrective measures can be taken. An example of a single run task which may be started upon a report of timeout is one which will sample the individual clocks of each processor of the system to detect any significant disagreement. This task may in turn report the findings to the system for further corrective actions.

The overall structure of the kernel has now been presented. A corresponding task state transition diagram is given in Figure 5.2. Note that the states of the URV RTMOS tasks resemble those available in the commercial packages surveyed above.

One final topic in the context of the kernel merits discussion. Task code is static, ROMable, and reentrant. To activate a task or instantiate multiple copies of a task, task PCB's are used. These structures form the run-time representation of tasks. A task PCB contains a task's context when it is not running, its stack, queue linkage pointers, status flags, timeout information, and a pointer to the task code. To activate a task, a PCB is loaded with the appropriate initial task data and linked to the appropriate queue. To activate multiple tasks, multiple PCB's pointing to the same task code unit are created. Task code is either structured in an infinite loop, or contains a terminate kernel call. Appropriate processor release calls

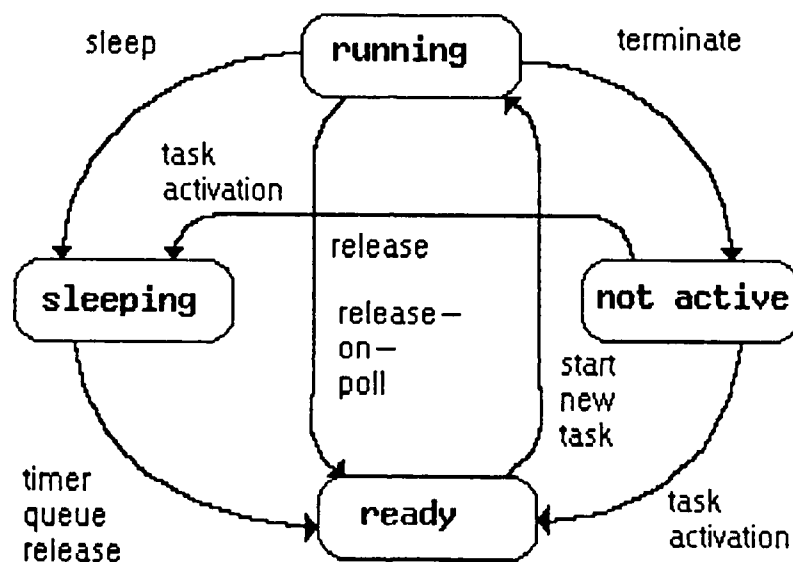


Figure 5.2

are either programmed in; or time slicing is enabled, allowing the kernel to automatically remove the task at periodic intervals. Examples of tasks can be found in the source code listing in Appendix (A3). Their associated PCB's are given with the source code for RTMOS in Appendix (A2).

(5.6) Remaining RTMOS Features

The kernel is just one part of the URV RTMOS. Figure 5.3 shows a layered representation of the RTMOS, its features, and its relationship to applications tasks. The only part of the RTMOS not addressed to this point is the system tasks which it employs. System tasks are different than applications tasks in that they are basically a part of the RTMOS, are always resident in each processor, and may have knowledge of their own processor. The reason why these are discussed last is that only a couple of system tasks have been designed and used in the URV multiprocessor avionics prototype. The first system task reports system errors to a terminal for diagnostic purposes. This task is initiated by the report-to-system system call. Because of the simplistic nature of the task, it will not be discussed in further detail.

Another system task implemented in the prototype provides an important interprocessor synchronization function. Without this capability, the individual processor clocks can skew. Since these clocks determine task release times from timer queues, skewing can create producer/consumer timing problems, including exchange timeouts. To prevent this, a periodic system task is

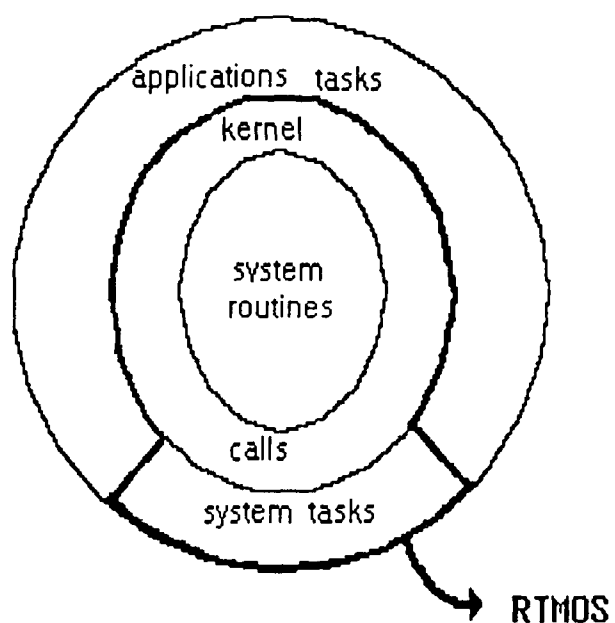


Figure 5.3

assigned to each processor to resynchronize the clocks of the system. For the purposes of the prototype, a simple algorithm is used. In this algorithm, each processor (actually the synchronization task operating on the processor) waits until all processors report to a table in shared memory, at which time the clocks are set identically. All of these synchronization tasks have the same timer queue release times and are the first tasks in each processor to be performed. If the tasks are scheduled appropriately, skewing will be controlled by reorienting the clocks at periodic intervals.

As is often the case, the simple solution is not the best. The above algorithm has a weakness in that the operation of the system is delayed until synchronization is completed. Also, the above algorithm does not account for the possibility of failed processors; a situation which will deadlock the system. A better algorithm would have each processor maintain its clock value or a copy of its clock value in its own section of the distributed shared memory. Resynchronizing a processor's clock would be the job of that processor alone, with no interaction with or waiting for another processor. The resynchronization would involve looking at all the clocks and resetting the local clock according to a consensus algorithm. This method is more complicated than it at first seems. More work remains in this area, however. The original algorithm given has sufficed for the first phase prototype.

Other potential system tasks, not designed for this first phase prototype, warrant discussion. The task

assignment for the current prototype is static; in other words, at initialization, the tasks are assigned to processors and loaded into queues. This assignment is never changed throughout the runtime of the system. However, a more practical scenerio would have the system configure the task assignment according to available processing resources and reconfigure the tasks in the event of a detected failure. The processes of task assignment, failure detection, and reconfiguration, therefore, comprise an important part of the system task area. These tasks will be discussed in the following paragraphs in terms of their theory of operation. Other system tasks are possible, but further work on the URV RTMOS is needed to determine the requirements.

Task assignment can be carried out in many ways. One technique which has been proven is the CRMMFCS system of self-checks and volunteering [4]. Three basic steps are involved. Upon notification of a task assignment cycle, a task on each processor performs a brief self-check on its processor. The second step involves reporting the results (health status) to a table in the shared memory. The third step, which takes place after a suitable delay to let all processors report, has a task check the status of the table, count the number of healthy processors, and choose a set of tasks. Various acceptable techniques exist for this last step; as such, no further discussion is required.

Failure detection tasks can take many forms also. Low priority self-checks can be used for "background" built-in-test. The clock checking task put into the system upon timeout (described previously) is another example. These

self-check tasks can be of single run type, timer queue type, or low priority ready-queue-only type. The results of the failure detection can result in a variety of responses. The timeout task above could resynchronize the clocks if the current clock value of each processor is kept in shared memory. Perhaps a reconfiguration could be triggered by interrupts or special command words monitored regularly by the individual kernels.

Reconfiguration itself is a variant on the task assignment task(s). Reconfiguration involves recognizing the request, confirming the validity of the request, and carrying out the request. This last step could be accomplished with a task assignment cycle running concurrent with the tasks still executing in the system. Of course, these system tasks would be of higher priority than applications tasks. Note that we may allow tasks currently in the system to continue to execute during the reconfiguration process. The tasks in the system at the time of the reconfiguration, however, are marked such that they are not reloaded into the system queues upon their release of the processor (except for polling or time slice). In this way, new tasks assigned can be loaded into the queues while the "old" set is finishing to allow a smooth and uninterrupted transition.

(6) Laboratory Approach

As discussed in the previous chapters, the hardware used in this initial development stage consists primarily of off-the-shelf purchased items. Some modifications and additions were made. These enhancements will be discussed in this chapter. Similarly, the operating system and hardware test software development began from off-the-shelf software. A debugger monitor, written by the author for a previous project, was used for initial hardware checkout and provided the basis for early kernel testing. This software, as well as the development of kernel itself, will be discussed in this chapter. In addition, the applications tasks, I/O software, and simulation set-up will be addressed.

(6.1) Processor Board

The four processor boards purchased required only minor modifications. As discussed previously, each board's dual port RAM can be addressed uniquely by requests over the VME bus. This required modifications to be made to the dual port address decoder implemented in a Programmable Logic Array (PLA) chip. Once this change was made, the shared memory segments of the four boards were addressed beginning at 80000H, 100000H, 180000H, and E80000H.

To check out the boards, the above mentioned debugger monitor was modified to match the new address map and utilize the MC68681 DUART chip for character I/O. This monitor software provides some basic commands such as dump

memory, examine and modify memory, fill memory, display registers, set breakpoints, and begin user program execution. A RAM test command, however, was most important. This command allowed the new decoding PLA's to be tested, and provided a means to test remote VME bus accesses. Of particular interest was a test of bus loading. This test was performed with simultaneous RAM tests being executed across the VME bus by multiple processors. The test provided a measure of VME performance in a multiprocessor environment in a near worst case loading situation. The results of this test are given in Chapter 7.

To determine the DUART's capabilities to act as the generator of the 'tick' function, a modification was made to the monitor in the initial stages of testing. The DUART was programmed to provide a once-per-millisecond interrupt via a countdown timer. The interrupt routine increments a counter in memory. Two commands were added to verify the operation of the routines. One gives the time in minutes and seconds since monitor start-up and the other gives the raw counter value.

(6.2) Kernel Development

The decision was made at the start of the software development process to incrementally design and test the functions and data structures of the kernel. First, the basic ready queue and queue access routines were developed. Next, a set of test tasks and the requisite task communication macros were added. This collection provided the basis for the first multitasking tests. When this stage was verified, the timer queue and queue access routines were

added and tested. This iterative process was followed until all the features of the kernel were incorporated and tested. After the verification of the kernel, the applications tasks were designed and implemented.

The design of the kernel functions and data structures was fairly straightforward. Typically, however, the testing of such features can be a difficult and lengthy process. The decision was made to develop test tasks which could visually demonstrate the correct operation of the kernel. Concurrent testing of tasks and the kernel was not desired. As such, the debugger monitor discussed above was used as the basis of test tasks for the kernel. The reasons are simple. The individual commands of the monitor were previously verified and, with little modification, were convertible to tasks. Each of the commands chosen are user interactive or provide visual evidence of operation.

In the initial task set, five commands were implemented: dump memory, examine memory, fill memory, show time, and show counter. To complete the set, the mainline command interpreter, character input, and character output routines were also converted to tasks. Character I/O was handled by utilizing character input and output queues protected by special kernel functions. Proper operation of the ready queue and associated routines was demonstrated by utilizing two processors; one with the command line interpreter and I/O tasks, and the other with the monitor command tasks and I/O tasks. Multiple monitor commands, such as dump and fill, were executed simultaneously, with results visually demonstrated.

To test the timer queue operations, command tasks were set up to operate at specified periodic rates. For example, the examine task was set up to execute once every thirty seconds. Obviously, this could be checked by forcing the initiation of a signal to the examine task data structure (via p.poll) with the command line interpreter task. The examine task, although signalled, did not respond until released from the timer queue at the end of its thirty second wait. Again, test results were visual.

Similarly, communications timeouts and time slicing were verified using these tasks. Although not all kernel problems were detected using these monitor tasks, most of the complicated "bugs" in the data structure routines were eliminated.

(6.3) Floating Point Hardware

Before the applications software could be designed and tested, the addition of the floating point coprocessor, MC68881, was required. This coprocessor was not included on the purchased boards. The side connector on the processor board, however, allowed for the addition of a wirewrap extension board. A schematic of the added hardware circuits is given in Appendix (A4).

The MC68881 was developed primarily to be used with the MC68020 processor. The coprocessor, however, can be used as a peripheral device for other microprocessors [26]. A simple sequence of software instructions can be used to coordinate transfers of commands and operands between the MC68000 and MC68881 [27].

To test the developed wirewrap circuits, the

nonmultitasking version of the debugger monitor was again used. Additional commands were added to perform transfers to and from the coprocessor registers, floating point additions and multiplications, and integer multiplications and additions. A single coprocessor register was used as an accumulator. Once these single operation routines verified the operation of the coprocessor, the applications software was designed and tested.

The MC68881 handles three types of floating point data formats. Single precision is 32 bits wide (1 sign bit/8 bit exponent/23 bit mantissa), double precision is 64 bits wide (1/11/52), and extended precision is 80 bits wide (1/15/64). In all internal operations and registers of the coprocessor, extended precision format is used. For the purposes of the applications tasks to be discussed below, single precision was determined to be sufficient. As such, all floating point numbers stored in main memory or in MC68000 registers are kept as 32 bit single precision values. Format conversions are performed automatically by the coprocessor during operand transfers.

(6.4) Applications Software Development

The application chosen to demonstrate the prototype multiprocessor system was a control mixer for reconfiguration of control laws of the URV in the event of control surface failure. In short, this concept modifies the control surface gain matrix to offset the effect of a failed surface by distributing control authority to the other control surfaces of the aircraft. In effect, the remaining

surfaces act to compensate for the loss. Much control and mathematical theory has gone into this research; however, the theory is outside the scope of this project. Interested readers are directed to the references for more detailed information.

As defined in [28], the linearized continuous aircraft state equations of the unimpaired aircraft are given by

$$\dot{x}(t) = A x(t) + B_o d(t)$$

where $x(t)$ is the aircraft state vector, $d(t)$ is the aircraft control surface deflection vector, and B_o is the control effectiveness matrix. Also,

$$d(t) = K_o u(t)$$

where $u(t)$ is a pilot plus flight control system (FCS) input vector and K_o is the control mixer gain matrix. Expanding the above equation, we get

$$\dot{x}(t) = A x(t) + B_o K_o u(t).$$

As will be discussed below, a failure changes the makeup of B_o . In order to keep the aircraft model the same (ie: tolerate the failure), the quantity $(B_o K_o)$ must remain constant. As a result, the K_o matrix must be adjusted to offset the changes in B_o . If we use o subscripts to signify matrices of the unimpaired aircraft, and i subscripts to signify matrices of the impaired aircraft, we get

$$B_i K_i = B_o K_o.$$

To solve for K_i

$$K_i = B_i^{-1} B_o K_o.$$

However, B_i may not be a square matrix, in which case it is not invertible. As such, we use the fact that a matrix multiplied by its transpose results in a square matrix. As shown in [28],

$$K_i = (B_i' B_i)^{-1} B_i' B_o K_o. \quad (6.1)$$

The derivation of this equation assumes that B_i is an $m \times n$ matrix, where m (number of state equations) is greater than n (number of control surfaces).

For the purposes of the applications functions of the prototype test, B_o is a 5×5 matrix, where each column corresponds to a control surface on the URV. It is assumed that a control surface fails in the center and locked position. As such, all entries in the column corresponding to the failed surface go to zero. This resultant matrix is B_i .

If B_i were left in this form, we would be unable to compute K_i since the quantity $B_i' B_i$ is singular, and, therefore, is not invertible. As a result, we convert B_i such that $B_i' B_i$ becomes potentially nonsingular by removing the zero column. This conversion leaves B_i as a 5×4 matrix. With K_o sized at 5×3 , K_i computes into a 4×3 matrix. Note that each row of the K_i matrix corresponds to a control surface of the aircraft. As such, a row of zeros can be reinserted into the row corresponding to the lost

surface. The resultant gain matrix, K_i is a 5×3 matrix with the gains of the good control surfaces adjusted to compensate for the loss of the failed surface. This derivation can be extended to multiple failed surfaces.

For the demonstration of the prototype, the job to be performed was the real time computation of the K_i matrix in response to failed surfaces. Previous tests on the TN17 aircraft were run using precomputed control mixer gain matrices. The matrices were derived on the ground and placed into the ROM memory. The 8061-based autopilot would simply select the correct gain matrix given the failure induced. Note that the "failure" is really a control panel switch which signals to the autopilot which surface to "fail".

Real time computation of the control mixer gain matrix is too intensive of a job for the 8061 to handle alone. As such, the demonstration of the multiprocessor prototype has been designed to compute K_i continuously on the basis of failure information supplied by the 8061. For the purposes of this demonstration, the basic autopilot functions have been left in the 8061 processor. The K_i function in equation (6.1) is broken into a set of tasks, much of which can be run in parallel. These tasks are triggered by a preceding task which samples the failure data supplied by the 8061. The last task in the computation of K_i triggers a following task which converts the computed matrix to the form used by the 8061 and stores it in dual port memory where the 8061 can access it. This real time computation is transparent to the 8061 in that the 8061 simply uses the gain matrix currently stored in memory. By keeping the autopilot

function in the 8061, the demonstration of the prototype is simplified. The modifications to the existing autopilot are minor, and fewer 68000-based tasks are required. These assumptions do not detract from the intent to demonstrate the capabilities of the prototype.

(6.5) Test Configuration

Figure 6.1 gives the system test configuration used for the demonstration of the multiprocessor prototype. One aspect of the set-up requires special note. The 8061 processor hardware is connected to one of the MC68000 processors through dual ported RAM on the attached wirewrap board. This set-up is not the ultimate configuration of the TN21 avionics system. As discussed in chapter 4, the 8061 in the system will be interfaced through the VME bus. To simplify the demonstration hardware, the 8061 section was connected to one of the MC68000's to eliminate the need for VME interface hardware to be developed. The dual port interface, in contrast, is simplistic and makes for effective demonstration of the concepts of the prototype. The two extra tasks described above were assigned to the MC68000 interfaced to the 8061 to make this deviation somewhat transparent. A later phase of the development of the TN21 system will include the implementation of the proper interface and software.

(6.6) Parallelization of the Ki Function

To divide a function, such as equation (6.1), into tasks for a multiprocessor, one must first identify the areas of potential parallelism and the areas of strict

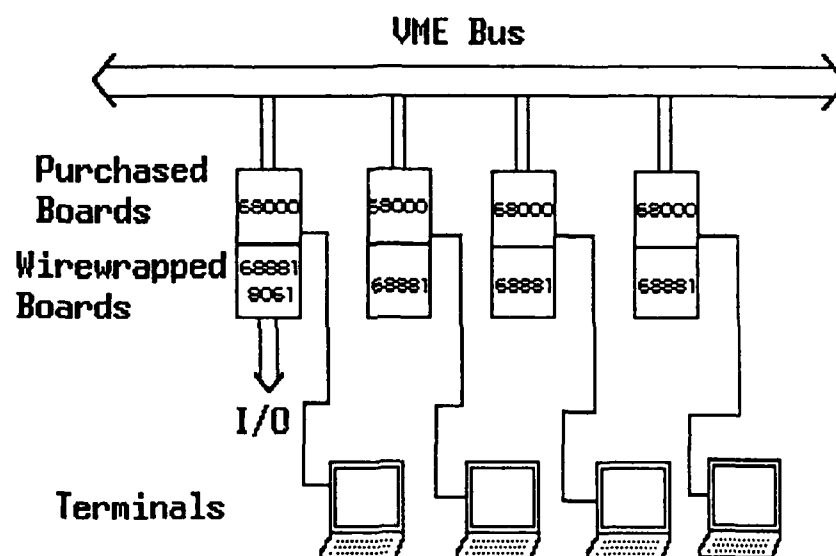


Figure 6.1

serial dependancies. First, let us examine a sequential algorithm for the computation of K_i .

- (1) Take the transpose of B_i ($= B_i'$)
- (2) Multiply B_i' by B_i ($= A$)
- (3) Take the inverse of A ($= C$)
- (4) Multiply B_i' by $B_o K_o$ ($= D$)
- (5) Multiply C by D ($= K_i$)

Note that $B_o K_o$ is a static entity, and as such can be precomputed.

First analysis of the above algorithm shows only one area of parallelism. Step (4) is independent of steps (2) and (3) and as such can be performed concurrently. Other less apparent areas of parallelism exist if the steps of the algorithm are defined in more detail. For example, step (3) involves a matrix inversion which is comprised of many substeps.

A basic method for computing a matrix inverse is given by

$$A^{-1} = \text{adj } A / \det A$$

where $\det A$ is the determinant of A and $\text{adj } A$ is the adjoint of matrix A and is defined as

$$\text{adj } A = (\text{cof } A)'$$

The cofactor of A , $\text{cof } A$, is defined as the matrix whose row

i and column j entry is given by

$$\text{cof } A(i,j) = \text{minor}(A(i,j)) * (-1)^{i+j}.$$

The $\text{minor}(A(i,j))$ is the determinant of the submatrix obtained by eliminating the i th row and j th column of A . This process of matrix inversion is only applicable to square matrices that are nonsingular. Given that $B_i' B_i$ is always a square matrix, the first requirement is satisfied. The second requirement that the matrix need be nonsingular is less certain.

Other techniques exist to compute the pseudoinverse of matrices in cases where the matrix in question may be singular. Examples include the CROUT [29] and singular value decomposition [30] techniques. These methods are certainly preferred for use in actual flight systems performing inversions on state matrices, since the values in these matrices are uncertain and may contain many very small or zero valued items. In fact, the assumption that the column of B_i corresponding to the failed surface goes entirely to zero makes B_i singular in its unmodified form. This assumption forced the removal of that column of B_i in order to use the above basic inversion method. The reason why the basic technique is used in the prototype demonstration is that it is highly computationally intensive, and as such, is a suitable test of the capabilities of the prototype. Also, the basic technique, in contrast to the pseudoinverse techniques, is straightforward in implementation, thereby simplifying the prototype demonstration effort.

Condensing the basic inversion function, we get

$$A^{-1} = (\text{cof } A)' / \det A.$$

This function can be accomplished by the following sequential algorithm replacing step (3) from above:

- (3a) Compute $\det A$
- (3b) For each $A(i,j)$
- (3c) Compute $E = \det (\text{minor} (A(i,j)))$
- (3d) If $i+j$ is odd then $E = -E$
- (3e) $E = E / \det A$
- (3f) Store E at $C(j,i)$
- (3g) Next $A(i,j)$

Obviously, steps (3b) to (3g) involve independent computations for each element of A . Each element can, therefore, be handled concurrently.

In as much as the matrix inversion step can be broken into smaller, potentially parallel substeps, so can the matrix multiplications and determinants. These functions also involve independent threads of computation which can increase parallelism. However, given the matrix sizes discussed above, it does not seem beneficial to enforce parallelism at this fine a level. The computations in parallel should be as intensive, or course grained, as to justify the added intertask communications and kernel overhead costs. Also, the number of processors envisioned (six to eight maximum) and the concentration of parallel tasks already in the execution regions of the functions

lessens the impact of the increased parallelism. As a result, the breakdown of the Ki algorithm has been performed to a sufficient level to define task units. Figure 6.2 graphically shows the algorithm with its sequential and parallel sections.

Various methods exist to convert the above algorithm to tasks and communications. Figures 6.3 and 6.4 show two such methods. In Figure 6.3, the tasks operate in a dataflow-like manner. A task executes only after receiving inputs or a signal to proceed. Each task is of the form:

- (1) Receive inputs
- (2) Compute results
- (3) Send outputs.

The implementation in Figure 6.4 is similar, but takes the appearance of a main program and subroutines. The KI task controls the ordering of calls to the four tasks it communicates with. In turn, the tasks that the C task communicates with are like subprograms local to the C task. Data movement is bidirectional, unlike in Figure 6.3 where it is unidirectional. The numbering in Figure 6.4 represents the sequencing order of the "calls". "Calls" of equal sequence number are concurrent. Task communications of this type are similar to those termed remote procedure calls in other literature [31].

The remote-procedure-call-like format has been utilized in the TN21 prototype system due to the similarities to conventional programming. The following sequence of macro

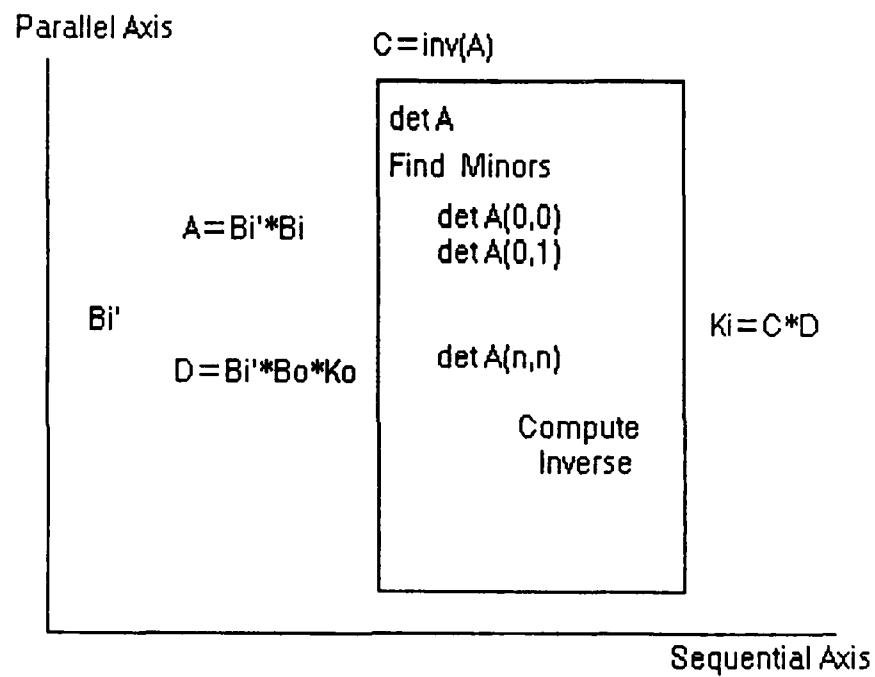


Figure 6.2

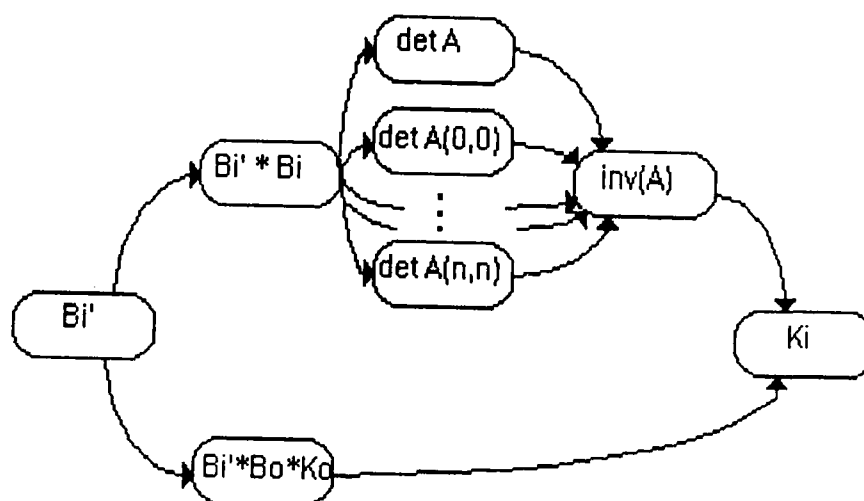


Figure 6.3

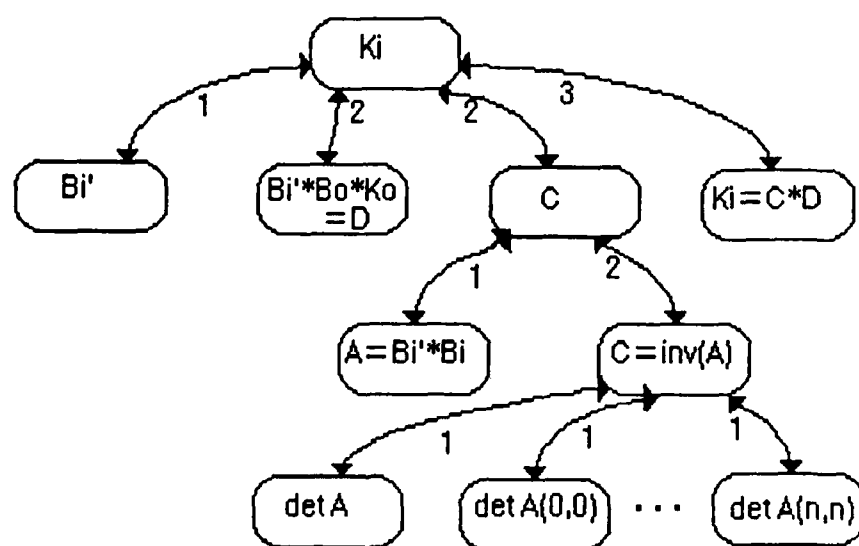


Figure 6.4

instructions implement the KI task "mainline" routine from Figure 6.4:

```

p.poll    transstart,KIdata
c.poll    transend,d0-d1,KIdata
p.poll    Cstart,KIdata
p.poll    m453start,KIdata
c.poll    m453end,d0-d1,KIdata
c.poll    Cend,d0-d1,KIdata
p.poll    m443start,KIdata
c.poll    m443end,d0-d1,KIdata

```

A "procedure call" is implemented by a corresponding p.poll/c.poll pair of macro instructions. Parallel "procedure calls" are made by multiple p.poll instructions in sequence. Recall that only c.poll instructions wait for data exchanges. As a result, the initiation of "calls" to C and m453 above allow parallel operation of remote procedure call tasks.

Some parts of the algorithm have strict serial dependancies. For example, the transpose of B_i has to be performed before all other parts of the algorithm. As Figure 6.2 graphically demonstrates, the fastest the algorithm can be completed is roughly the sum total of the times to do a transpose, two matrix multiplications, and a determinant. However, this alternative is much better than the sequential cost of a transpose, three matrix multiplications, and seventeen determinants (sixteen of which are a part of cofactor derivations). Chapter 7 will discuss the impact of parallel processing on the algorithm.

(6.7) Allocation Of Tasks Onto Multiple Processors

The following is the list of tasks used in the Ki

computation:

```

KI           : "mainline"
trans        : take transpose of Bi
m453         : compute  $D = B_i' B_o K_o$ 
C            : initiate computation of  $C = (B_i' B_i)$ 
m454         : compute  $A = B_i' B_i$ 
Ainv         : initiate determinants, compute A
det4         : 4X4 determinant of A
det30-det315: 3X3 determinants of minor(  $A(i,j)$  )
m443         : compute  $K_i = C D$ 

```

Each task is given equal priority for the ready queue. The most efficient operation can be accomplished through proper ordering on the timer queue, with release times assigned accordingly.

The assignment of tasks to a single processor is somewhat simple. The proper ordering on the timer queue is the same as the order of execution in the serial algorithm. As such, the above ordering works best for the single processor case. Timer queue release times have little effect on the overall performance since trans, m453, m454, m443, and the determinant tasks will all execute until completion once initiated. As such, little polling will occur.

In the case of multiple processors, the situation is not as simple. Care must be taken to evenly distribute the processing load over the processors, taking execution time into account. The important idea is to keep potentially parallel tasks, such as m454 and m453, operating concurrently. The placement of the c.poll and p.poll instructions, as demonstrated above for the KI task, defines to a large extent this parallelism. Two other basic rules-of-thumb are as follows. Obviously, the tasks that can execute in parallel should be separated onto different

processors to allow for true concurrent operation. To insure that the separated tasks can operate efficiently in parallel, a second rule-of-thumb involves the proper use of timer queue release times to limit delays in initiation of tasks due to polling. For example, the determinant tasks (seventeen in number) cannot possibly start until the transpose and first two matrix multiplications are finished. If the determinant tasks' PCB's are loaded into the ready queue at the same time as the other tasks', the mass of tasks involved in polling could cause delays in tasks getting started, thereby degrading the effect of parallelism.

Figures 6.5 and 6.6 give the timer queue orderings for the processors in the two and three processor configurations respectively. From these, one can see that the main effect of parallelism as more processors are added is in the determinate tasks. Obviously, the best performance to be expected would involve the use of seventeen processors; however, the small performance gain would not justify the hardware costs. The next chapter will address performance measures determined for this algorithm.

(6.8) 8061 Hardware Design

No new schematics were required for the 8061 circuit used. The existing autopilot design provided all the required memory and I/O interfaces. The only modification needed was a change in the type of RAM memory chips used. Given that the 68000-to-8061 interface dual port RAM was already specified for use, these chips were substituted in

2 Processors

Processor 1

KI
m453
det38
det39
det310
det311
det312
det313
det314
det315
m443

Processor 2

trans
C
m454
Ainv
det4
det30
det31
det32
det33
det34
det35
det36
det37

Figure 6.5

3 Processors

| Processor 1 | Processor 2 | Processor3 |
|-------------|-------------|------------|
| KI | trans | m453 |
| C | m454 | det4 |
| Ainv | det30 | det36 |
| det312 | det31 | det37 |
| det313 | det32 | det38 |
| det314 | det33 | det39 |
| det315 | det34 | det310 |
| | det35 | det311 |
| | m443 | |

Figure 6.6

the schematic design for the static RAM chips previously there. The 50-pin connector used for I/O interface was left to allow direct connection to a simulation-interface box used in previous URV autopilot hardware-in-the-loop simulations. This configuration allows the prototype to be "plugged" into the simulation facilities identically as in previous tests. The resultant circuit occupies the attached wirewrap board space as shown in Figure 6.7.

(6.9) 8061 Software Modifications

As was mentioned previously, the existing autopilot functions were left in the 8061 processor to simplify software changes. Some changes were required, however, to link the 68000-multiprocessor-based control mixer software with the basic autopilot. The previous 8061 control mixer software, consisting of a matrix selection algorithm based upon failure number, was removed and replaced with a routine to write the failure number into the dual port memory and assign the dual port gain matrix address for all failure cases. As stated before, the 68000 multiprocessor will use the failure information to compute and store the appropriate gain matrix for the 8061 autopilot to use.

(6.10) Hardware-in-the-Loop Simulation Configuration

The configuration of the simulation tests is given in Figure 6.8. The prototype hardware is connected to a device which contains servos corresponding to each of the control surfaces on the URV. The prototype hardware commands these servos directly. The simulation computer detects the servo

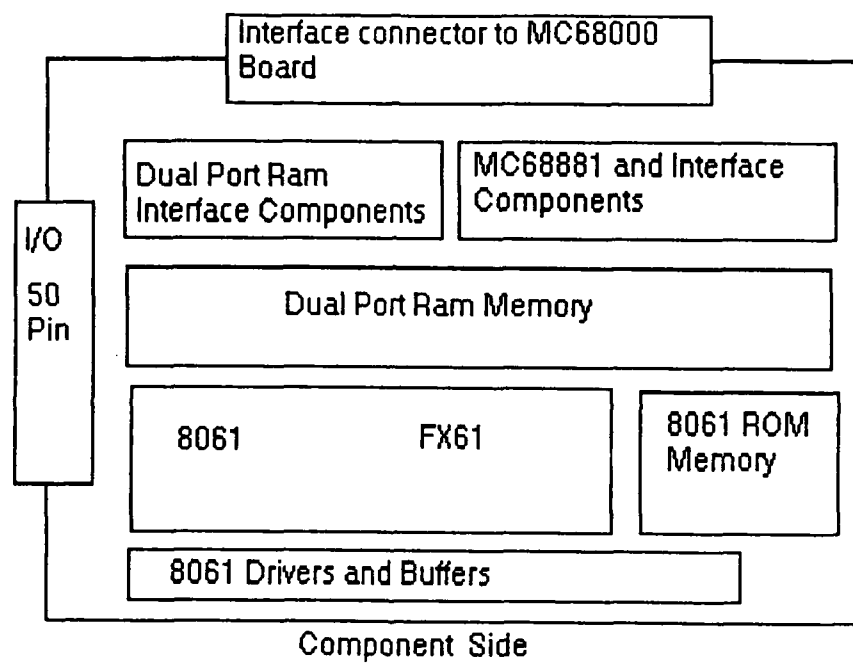


Figure 6.7

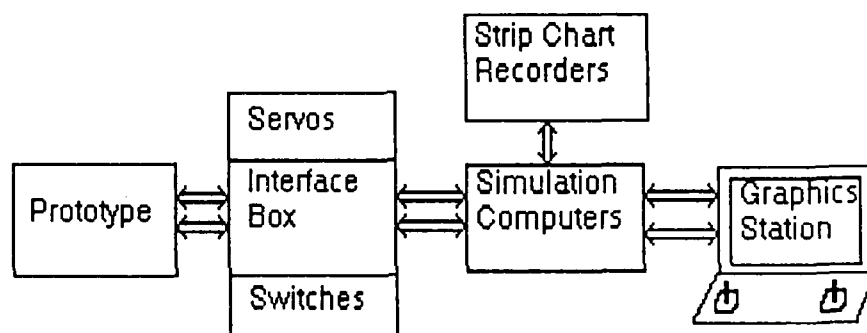


Figure 6.8

movement as input to the airframe simulation. The graphic display station also provides inputs to the simulator computer in the form of pilot commands. Rudder, elevator, aileron, and throttle command input devices are located at this station. The simulation computer returns sensor data to the prototype hardware through the servo interface box and aircraft state data to the graphic display station.

Surface failure is accomplished via switches on the servo interface box. A failure switch exists for each of the URV control surfaces. The switch settings are converted by the servo interface box to inputs to the 8061 circuit in the prototype. Visual confirmation of the failure can be made during a test run by monitoring the servo corresponding to the failed surface. Once failed, the appropriate servo ceases to move.

With this configuration, the URV multiprocessor prototype can be tested in a real time environment. The graphic display provides simulated artificial horizon, attitude, speed, climb rate, angle of attack, and side slip angle indicator devices. With these, the "test pilot" can verify the operation of the hardware under test with an environment similar to an actual URV flight. This is important, especially when evaluating performance of the control mixer under failure conditions.

(7) Prototype Development and Demonstration Results and Findings

The purpose for producing and demonstrating a prototype, such as the one in this research effort, is to prove the anticipated benefits, identify problem areas, establish areas of further research, and provide data for use in later system development stages. The development of the TN21 prototype multiprocessor system realized these goals. The anticipated benefit of high throughput capability was demonstrated through the significant decrease in time to compute a complex arithmetic function with only a few processors. Several problem areas have been discovered and corrected, including some in the applications functions area. Data has been collected on hardware and software performance. These aspects will be discussed in this chapter. Those areas requiring further research will be identified in Chapter 8.

(7.1) VME Performance and Bus Loading

Chapter 4 addressed the concerns anticipated in the use of a single bus. The claim made was that, with sufficient bus bandwidth and limited numbers of processors connected, bus loading would not become significant. To back up this claim, analyses were made for the VME bus, both in the theoretical and physical cases.

(7.1.1.1) VME Access Options

A discussion of VME access options [32] is first required. The system bus controller on a VME bus can implement a priority-based or round-robin-based arbitration scheme. The VME bus has four priority levels. Each potential bus master is assigned to one level. The assignment of a bus in the priority mode of operation is based upon those priority levels. A processor with a higher priority level is assigned the bus before a processor with a lower priority. The round-robin mode, in contrast, offers a scheme based on fairness. The assignment of the bus is rotated between priority levels. As such, no given priority level can be "starved" from bus access. The VME boards purchased provide for the selection of either option; however, in the analyses to follow, round-robin is assumed.

Once a processor is granted the bus, it may either hold the bus for the entire time required, or only for a single bus transfer, depending upon the release option. In the release-when-done (RWD) option, the bus master has control of the bus until it has completed all of its transfers. In the release-on-request (ROR) option, another bus master is assigned after the current transfer, if a request for the bus is made. Minimal access latency is achieved with the ROR option. In the purchased boards and the analyses to follow, this option is used.

Bus arbitration may either be handled during the current bus cycle or after the bus cycle. Obviously, maximum bus bandwidth is achieved with concurrent bus usage and arbitration. This option is used on the hardware and in the analyses.

(7.1.2) VME Bus Access Latency Effects (Theory)

If n processors are to use a single bus and if true fairness is assumed, a processor may have to wait up to $n-1$ bus transfer cycles to get access. A basic MC68000 memory cycle takes four clock cycles to complete, if no wait states are applied. At a processor clock speed of 10 MHz, 2.5 million transfers per second can be made. This translates to a memory access time of 400 nanoseconds (nsec).

The VME bus can be viewed as an extension to the MC68000 bus. To account for signal propagation over the VME bus and VME bus arbitration, let us assume an additional clock cycle per transfer. This assumption increases the single transfer time to 500 nsec. If six processors are connected to the VME bus, the maximum bus access latency would be 2.5 microseconds (usec). This latency is on the order of the execution time of all MC68000 instructions at a processor clock speed of 10 MHz. If the ratio of non-bus accesses to bus accesses is sufficiently high, the added time for off-board memory cycles is not too significant.

Consider, however, a case where the ratio is not very large. A block move loop is a near worst case example.

```

loop1      move.w    d0,(a0)
           cmpa.l    a0,a1
           bne       loop1

```

The above loop takes 2.4 usec at 10 MHz for local memory accesses. If we use the latency time of 2.5 usec above and assume the worst case of always suffering the maximum latency, the loop will take approximately twice as long over

the VME bus as for local memory accesses. However, note that the latency assumptions were based upon six processors competing for the bus. Even if all six experienced a simultaneous 100 percent increase in computation time using near worst case loops such as the one above, the effective system parallel speed up is a factor of three. Translating down to a three processor contention scenario, we would see a 50 percent increase in computation time and a speed up factor of 2.25.

(7.1.3) VME Bus Access Latency Effects (Measured)

To further illustrate the effects of bus access latency under the assumptions of the research program, a test was run on the project hardware to measure actual contention effects. The RAM test function included in the debugger monitor utilizes tight loops similar to the one given above. By installing the monitor code on each of multiple processor boards and utilizing the RAM test function on each to access off-board memory, a scenerio like the one above can be tested. Note that the assumption of always suffering the worst case latency will not apply here.

Utilizing two processors, a RAM test covering 32 Kbytes of memory took 55.6 seconds under contention conditions. The same test took 48.2 seconds without contention. This translates to approximately a 15 percent increase in computation time. Performing the same test with three contending processors resulted in an execution time of 59.8 seconds (24 percent increase). Note that the measured results are considerably less than the calculated case in the previous section; testimony to the fact that the worst

case latency is not frequently realized. The three processors can achieve a 2.4 factor of speed up in contention traffic similar to the RAM tests. A six processor configuration would certainly achieve a much better speed up factor.

(7.1.4) VME Bus Access Latency Effects (Conclusions)

The result of these simplistic analyses is that the VME bus is sufficient for the TN21 multiprocessor system, given the assumption of six to eight processors maximum. Near worst case loops, such as the one given previously, will not typically appear in actual applications code, simultaneously on multiple processors. Block transfer loops to shared memory do appear in the p.poll and c.poll routines. However, non-VME access to VME access ratios of 200:1 or greater are typical in the Ki computation routines (25 usec transfer time to 5 milliseconds (msec) computation time). As a result, contention is minimal and the concern of "bus bottleneck" is alleviated. As mentioned before, this analysis only applies to non-fine-grained computations. The applications on the TN21 multiprocessor are assumed to be coarse grained. Shared memory is used for data transfer and storage. Local memory is used for intermediate results, around which most computation time is spent.

(7.2) Execution Times for Kernel Routines

The performance of the multiprocessor/multitasking kernel is a key element in the overall system computation efficiency. Context switches were one such aspect of kernel

performance discussed in Chapter 5. Kernel performance relates directly to overhead, which is time not spent on applications functions.

(7.2.1) RTOS Comparisons

[16] gives some typical timings of RTOS's. In general, context switch times of 75 to 150 usec can be expected with these commercial products (processor clock speeds unknown). The times solely represent the time to switch tasks on the processor. Other timings, such as for calls for clock acquisition or mailbox utilization, are not given in the comparison table; however, typical values of 200 usec for service calls are mentioned in the text of the reference. In all fairness, the computation of such figures-of-merit are difficult, given the variable conditions present (number of tasks in the system, interrupts, etc). Some attempt will be made to quantify these for the URV RTMOS.

(7.2.2) RTMOS Timings

The following is a list of execution times for typical kernel service routines implemented as MC68000 TRAP's in the RTMOS.

| | |
|------------------|------------------|
| Exit | : 85.6 usec |
| Release-on-poll | : 92.8 usec |
| Sleep | : 91.4 + 9n usec |
| Report-to-system | : 38.6 usec |

where n = number of positions from the front of the timer queue where the task is placed

Note that the first three times comprise the basic context switch routines of the RTMOS. All are comparable to the

RTOS times referenced above.

Other kernel timings are more critical. For example, the p.poll and c.poll macros may include release-on-poll kernel calls, but also contain other areas of overhead. The p.poll macro takes $34.4 + 2.8x + 3.4y$ usec, where x is the number of times attempting the MC68000 test-and-set (TAS) instruction used for data structure mutual exclusion and y is the number of data words transferred. Typically, a p.poll will take less than 50 usec; however, if larger amounts of data (such as a matrix) are transferred, the time may increase to around 100 usec.

C.poll takes $37.2 + 2.8s + 100.4p + 2.8tp + 3.4q + 1.6r$ usec to complete, where s and t are counts of TAS executions, p is the number of times through the poll wait loop (with release-on-poll), q is the number of data words transferred, and r is the number of 32 bit registers saved. The minimum c.poll time, assuming no waiting on TAS instructions or polling variables, is 45 usec. More realistically, however, $s=t=2$ and $p=1$. If we assume $q=2$ and $r=8$, the time for c.poll is 168.4 usec. As with p.poll, if more data is transferred, the time will increase accordingly.

The tick function interrupt service routine (ISR) is another critical consideration in kernel performance. The following is a list of functions performed within tick and the times resulting:

| | |
|---------------------------|--------------------|
| Set up and clock update | : 15.2 usec |
| Get task from timer queue | : 3.6 + 41.6m usec |
| Time slice path | : 117.6 usec |
| No time slice path | : 20.4 usec |
| Interrupt handling | : 4.4 usec |

where m = number of PCB's released from the timer queue.

The total tick processing time is $140.8 + 41.6m$ usec if a time slice is performed and $43.6 + 41.6m$ if not. Timer queue staggering of tasks may be desirable to prevent long tick ISR times which can significantly delay applications tasks processing. One way or the other, the price in time eventually has to be paid.

(7.2.3) Implications of the RTMOS Timing Data

The kernel functions discussed above were not optimized for minimal execution time. Optimization is for final products, such as the RTOS products referenced, not for prototype systems. However, in all likelihood, very little significant improvement would be expected. At any rate, the times noted are comparable or better than those for commercial products.

One area where improvement could be made is in the timer queue storage routines. The current implementation searches the timer queue from front (least time to release) to rear to determine where the task should be placed. In actual use, most tasks would probably be placed closer to the end of the queue. As such, minimal search time would likely be achieved by beginning at the rear of the queue, moving forward.

The most critical implication of the timing data given involves the tick routine. A minimum tick execution would

NO-A194 886

A MULTIPROCESSOR AVIONICS SYSTEM FOR AN UNMANNED
RESEARCH VEHICLE(U) AIR FORCE WRIGHT AERONAUTICAL LABS
WRIGHT-PATTERSON AFB OH D B THOMPSON MAR 88

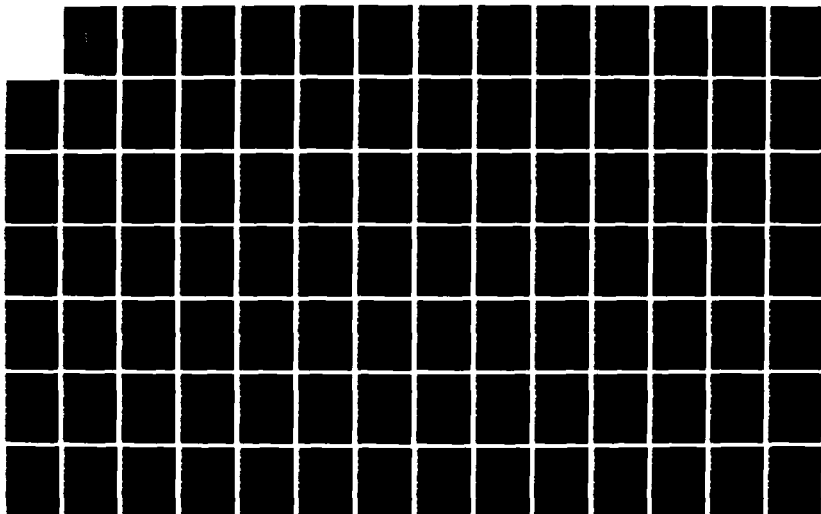
2/3

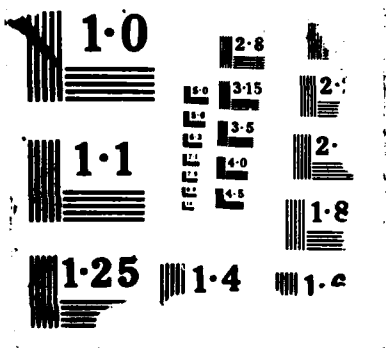
UNCLASSIFIED

AFWAL-TR-88-3883

F/G 12/6

NL





take 43.6 usec. Since a tick takes place once per millisecond, an automatic minimum overhead of 4.4 percent is realized. One time slice or a couple of releases from the timer queue would increase this to around 140 usec, or 14 percent. Although the means to improve the routine performance-wise have not yet been investigated, any optimization work on the kernel should be concentrated on the tick routine first. Interestingly enough, none of the RTOS's referenced give data on this timing characteristic, although all must have similar functions.

(7.3) Applications Computation Times

As with the kernel functions above, the applications tasks developed for the prototype were not optimized. For example, more extensive use of the floating point registers within the MC68881 floating point coprocessor as accumulators would have saved many processor-coprocessor data transfers. Optimization of these routines, however, would not have served to demonstrate the goals of this phase of development. Of more importance to these goals is the demonstration of significant speed up and minimal overhead when parallelism is applied to the problem.

(7.3.1) Sequential Limitations

As discussed in Chapter 6, Figure 6.3 demonstrates that the fastest time to compute the Ki function can be approximated by the sum total of the times to do the transpose, two matrix multiplications, and a 4X4 determinant. The times for these are given by

| | | |
|-------|---|------------|
| trans | : | .82 msec |
| m454 | : | 10.60 msec |
| m443 | : | 6.78 msec |
| det4 | : | 6.68 msec |

The sequential limitations of the algorithm, therefore, bound the computation time to no better than about 25 msec, no matter how much parallelism is applied.

(7.3.2) Measured Computation Times

Measurements of the actual times to compute the Ki algorithm were taken. Before the algorithm was parallelized, it was developed as a single processor, sequential program. This version took 59 msec to complete. The parallelized version required shared memory data exchanges via c.poll and p.poll instructions acting as remote procedure calls. Also, kernel overhead had some effect. The resulting times measured for the parallelized version using RTMOS are as follows:

| | | |
|--------------|---|---------|
| 1 processor | : | 69 msec |
| 2 processors | : | 47 msec |
| 3 processors | : | 36 msec |

As expected, the single processor version using the RTMOS suffers some overhead penalties (17 percent total). As parallelism is applied, however, the execution time drops accordingly.

As confirmation, Figure 7.1 demonstrates the division of tasks onto three processors. The fastest computation time for this division is the sum total of the transpose, two

| Processor 1 | Processor 2 | Processor 3 | |
|-------------|-------------|-------------|----------|
| KI | trans | | .82 |
| C | | | |
| Ainv | | | |
| | m454 | m453 | 10.60 |
| | | det4 | 6.68 |
| det312 | det30 | det36 | |
| det313 | det31 | det37 | |
| det314 | det32 | det38 | |
| det315 | det33 | det39 | 10.44 |
| | det34 | det310 | |
| | det35 | det311 | |
| | m443 | | 6.78 |
| | | Total | 35.32 ms |

Figure 7.1

matrix multiplications, one 4 X 4 determinant, and six 3 X 3 determinants. Each 3 X 3 determinant was measured to take 1.74 msec. The total, therefore, is 35.32 msec. This figure confirms the total algorithm computation time for three processors as given previously.

(7.3.3) Implications of Execution Time Data

One extra data point can be derived from the three processor timing confirmation in the previous section. Note that the difference between the total algorithm time and the sum total of the sequential parts is less than one millisecond. This translates to an overhead contribution of around 3 percent for context switches and tick handling. A previous section had predicted a minimum overhead contribution of 4.4 percent just for tick computations. These figures are fairly close, and serve to confirm expectations. They also indicate that context switching times are negligible. This finding is significant in that it demonstrates that polling does not contribute a high cost to the overall results.

The 17 percent increase in execution time from the sequential to single processor RTMOS versions can be attributed to two main factors. First, the overhead contribution of tick varies from a minimum of about 4 percent to as much as 33 percent during a time period when eight tasks are released from the timer queue. These peaks are rare, but still contribute to the overall overhead figures.

Secondly, the choice was made in the applications tasks

design stage to pass entire matrices through some p.poll and c.poll exchanges. In the sequential version, all accesses were local, so only addresses were passed between routines. The process of passing matrices is time consuming and could, in some cases, be replaced by matrix addresses. A side effect to this, however, is that intermediate computations would take place out of shared memory, rather than local memory.

(7.4) Timer Queue Utilization Timing

One hypothesis made in the early stages of the URV RTMOS concept development was that proper timer queue release time staggering would be required in order to limit the adverse effects of polling on performance. As demonstrated in the previous section, RTMOS polling has a minimal effect with coarse grained parallel computing. This finding would seem to indicate that staggering is less critical than expected. As confirmation, the applications problem was run with RTMOS on a single processor using and not using timer queue staggering. The result was an identical 69 msec execution time for both versions.

Although the lessened effect of polling can be attributed somewhat to this finding, another reason was discovered. After one iteration through the ready queue and processor, the timer queue becomes naturally staggered, regardless of its initial state. This effect is caused by the execution time of tasks on the processor and the subsequent delay in successor tasks beginning. Tasks are placed back on the timer queue as they complete; and, as such, have their next release time delayed accordingly. As a

result, only the first iteration is affected by the non-staggering order. All other iterations become naturally ordered and execute at optimal speed.

(7.5) Results of the Prototype Hardware-in-the-Loop Simulation

The prototype hardware, RTMOS, and control mixer applications tasks were tested under the simulation conditions described in Chapter 6. The simulation responded to the failures induced much as expected. The failure responses will be discussed below. In the development and initial tests of the applications tasks, errors in the Bo matrix were discovered. The problem and the corrections used will also be discussed. As mentioned in Chapter 6, the reader is directed to the references and other literature for further details concerning flight control, the URV aircraft, and the control mixer.

(7.5.1) Simulation Response to Failures

The control mixer model used was a five control surface model utilizing two elevators, two ailerons, and a rudder. A later model [33] utilizes two additional surfaces (flaps) on the URV which can be used to provide better response to failures, particularly in the roll axis. However, lack of sufficient information on this later model during the development stages prevented its use. Still, the earlier model provided for a sufficient computational load to test the multiprocessor.

The response to elevator failures was the best of all

cases. When one of the elevators was failed, the mixer provided double the authority to the other elevator in the pitch axis, and utilized the ailerons to assist in pitching the aircraft. The resultant response was suitable pitching control with a slight initial roll (corrected by the autopilot). The roll seemed to be induced by the aileron movement commanded by the mixer to assist the single elevator in pitching. Roll and yaw motion was not affected by an elevator failure, as expected.

Aileron failure response was not as good. As confirmed by off-line derivation of the gain matrix, the mixer response to an aileron failure was to effectively zero out the gain to the other aileron and the rudder in the roll axis so that only the elevators were used to roll the aircraft. The result was a sluggish roll response with significant downward pitching motion. This response was to be expected since the elevators have far more force in the pitch axis than in the roll axis. A better response would be to give more authority to the remaining aileron, such as was done for the remaining elevator in the failed elevator case discussed above. This observed response comes directly from the control mixer algorithm and URV model. The prototype calculations were confirmed by off-line matrix computations.

As discussed in [28], the rudder failure response, as computed by the control mixer, leads to unstable aircraft control. This is due to the fact that the elevators and ailerons do not have sufficient authority in the yaw axis to compensate for a rudder failure. As a result, the gains become excessive and the surfaces saturate. Again, this

response is a control mixer problem, not a prototype calculations problem.

(7.5.2) Bo Value Errors

In the development of the control mixer applications tasks, errors in the Bo matrix were discovered. Documentation on later URV control mixer work [33] was located, and an updated Bo was used. The primary difference between the original Bo matrix and the later version involved surface polarities. Differences existed between the surface direction assumptions made in the early stages of control mixer development and the actual aircraft set up. These differences were corrected in the later work. With the new Bo matrix installed, the expected responses, noted above, were observed.

(7.5.3) Potential Resolution Problems

One final observation on the control mixer applications used, and the 8061 autopilot, concerns the resolution of numbers calculated for the gain matrix. The numbers derived for the Ki gain matrix ranged from less than .001 to around 25. However, the conversion algorithm used to change the numbers to a form usable by the 8061 autopilot only allows for four bits of resolution to the right of the decimal point (fixed point format). As a result, the smallest gain magnitude greater than zero possible is .0625. Any gain magnitude smaller than this will be converted to zero. The conversion truncation is the reason why the remaining aileron and rudder gains, in the failed aileron case, are zero. Some authority is actually assigned in the floating

point calculations, but the gains are below the truncation threshold, and are lost in the conversion process. Further work may be needed in this area if further work on the control mixer is to be performed on the prototype.

(8) Conclusions

In order to make effective use of the new vehicle airframe being designed and constructed, and to provide high speed computing capabilities for embedded tests, AFWAL/FIGL has made the decision to develop a new avionics/control system incorporating a low cost multiprocessor architecture and software operating system. The effort is being performed in-house, utilizing years of multiprocessor system analysis, design, development, programming, and test experience. The first phase of this effort, described herein, has produced and demonstrated a prototype of this system. Multiple off-the-shelf MC68000 processor boards have been combined with a VME backplane bus and wirewrapped 8061 I/O circuitry and MC68881 coprocessors to form the hardware of the prototype. A real time multiprocessor/multitasking operating system (RTMOS) has been specified and developed to manage the parallel software units (tasks) of the system. Interprocessor communications protocols, and a simple methodology to use them to develop coarse grained parallel code, have also been developed.

The development of the URV multiprocessor avionics/control system is not yet finished. The next phase of development will bring the system closer to its completion by refining and enhancing the prototype in several areas. The 8061 I/O circuitry will be interfaced to the VME bus as originally specified. Additional MC68000 boards will be utilized to provide even more computing

power. The RTMOS will be fine-tuned for more efficient operation. Task assignment will be made dynamic, distributing the task load to the number of processors present. Research will be performed to specify and test a better multiprocessor clock synchronization scheme. The applications tasks will be enhanced to allow for a seven surface control mixer model that will be responsive to multiple surface failures.

In the longer term, the system hardware will need to be evaluated and modified for flight operation. An extensive verification procedure will also be required. Development and test of parallel software will have to be addressed from an applications programmer viewpoint. High order languages will need to be applied in order to make this software manageable. The impacts of changing airframe configurations on the control model and software will have to be assessed. A technique to allow changes to occur with minimal impact on software will be required.

In short, much work remains to be performed before the URV multiprocessor system is ready for actual implementation. Still, much has been accomplished; the foundation has been laid. Multiprocessor technology is beginning to see application in many areas. The low cost/risk URV research testbed is one area where significant payoffs can be realized.

Appendix A1
Macro Routines Listing


```

;.....
;see  MACRO Definitions
;.....

macro  exit
trap  #0
endm

macro  sleep
trap  #2
endm

macro  terminate
trap  #4
endm

; p.poll: producer poll macro
; format: p.poll pollvariable, localdata

macro  p.poll

    jsr  slice.off
    lea  "1",a0
    ; data structure addr=a0

    ppoll1"0"
    tas  semi(a0)
    bne.s ppoll1"0"
    ; P((a0))

    cmpi.w #cflag,flag(a0)
    beq.s pp.ok"0"
    ; flag = C?

    pp.fail"0"
    move.w #cflag,flag(a0)
    trap  #1
    bra  pp.end"0"
    ; if not then reset flag
    ; and inform OS

    pp.ok"0"
    move.w dcnt(a0),d0
    beq  pp.put0"0"
    movea.l a0,a1
    adda.l #dbeg,a1
    lea  "2",a2
    move.w (a2)+(a1)+
    subq.w #1,d0
    bne.s pp.putd"0"
    ; get data word count

    pp.put0"0"
    move.w #pflag,flag(a0)
    ; set flag = P

    pp.end"0"
    clr.b semi(a0)
    ; V((a0))

```

```

        jsr     slice.on
        endm

; p.pollm: producer poll macro (for grouped poll variables)
; format: p.pollm pollvariable,offset,localdata
macro    p.pollm
        jsr     slice.off
        lea     "1",a0
        adda.l  "2",a0
        ; data structure addr=a0

        tas     semi(a0)
        bne.s   ppollm1"0"
        ; P((a0))

        cmpi.w  #cflag,flag(a0)
        beq.s   pm.ok"0"
        ; flag = C?

        move.w  #cflag,flag(a0)
        trap    #1
        bra     pm.end"0"
        ; if not then reset flag
        ; and inform OS

        move.w  dcnt(a0),d0
        beq     pm.put0"0"
        movea.l a0,a1
        adda.l  #dbeg,a1
        movea.l "3",a2
        move.w  (a2)+,(a1)+
        subq.w  #1,d0
        bne.s   pm.putd"0"
        ; get data word count

        pm.put0"0"
        ; a1 = beg of data area
        ; a2 = beg of my data
        ; move data to data area

        pm.putd"0"
        move.w  #pflag,flag(a0)
        ; set flag = P

        pm.end"0"
        clr.b   semi(a0)
        jsr     slice.on
        endm

; c.poll: consumer poll macro
; format: c.poll pollvariable,registersused,localdata
macro    c.poll
        jsr     slice.off
        movem.l "2",- (a7)
        ; save registers

        lea     "1",a0
        ; a0 = data structure addr

```

```

cpoll1"0"      tas      semi(a0)      ; P((a0))
                bne.s    cpoll1"0"

clr.b    d0      ; timeout=false

cp.while"0"    cmpi.w    #pflag,flag(a0) ; while flag not = pflag
                beq.s    cp.endwh"0"
                cmpi.b    #0,d0          ; and not timeout
                bne.s    cp.endwh"0"

cp.do"0"       clr.b    semi(a0)      ; do V((a0))
                trap     #3            ; release-on-poll
                lea     #1",a0
                tas      semi(a0)      ; p((a0))
                bne     cp.do2"0"
                bra     cp.while"0"

cp.endwh"0"    cmpi.w    #pflag,flag(a0) ; if flag = P then
                bne     cp.time"0"
                move.w    #cflag,flag(a0) ; set flag = C
                bra     cp.getd"0"

cp.time"0"     move.w    #fflag,flag(a0) ; else set flag = F
                trap     #1            ; and report to system

cp.getd"0"     move.w    dcnt(a0),d0
                beq     cp.get0"0"
                movea.l    a0,a1
                adda.l    #dbeg,a1
                lea     #3",a2
                move.w    (a1)+,(a2)+
                subq.w    #1,d0
                bne.s    cp.getd2"0"

cp.get0"0"     clr.b    semi(a0)      ; V((a0))

movem.l    (a7)+,"2"      ; restore registers
jsr     slice.on
endm

; c.pollm: consumer poll macro (for grouped poll variables)
; format: c.pollm pollvariablebase,offset,registers,localdata
macro    c.pollm

```

```

jsr    slice.off
movem.l    "3", -(a7)
; save registers

lea     "1", a0
adda.l  "2", a0
; a0 = data structure addr

cpollm1"0"
tas     semi(a0)
bne.s   cpollm1"0"
; P((a0))

clr.b   d0
; timeout=false

cm.while"0"
cmpl.w  #pflag, flg(a0)
beq.s   cm.endwh"0"
; while flag not = pflag
cmpl.b  #0, d0
bne.s   cm.endwh"0"
; and not timeout

cm.do"0"
clr.b   semi(a0)
trap    #3
lea     "1", a0
adda.l  "2", a0
tas     semi(a0)
bne     cm.do2"0"
bra     cm.while"0"
; do V((a0))
; release-on-poll

cm.do2"0"
tas     semi(a0)
bne     cm.do2"0"
bra     cm.while"0"
; p((a0))

cm.endwh"0"
cmpl.w  #pflag, flg(a0)
bne     cm.time"0"
; if flag = P then
move.w  #cflag, flg(a0)
bra     cm.getd"0"
; set flag = C

cm.time"0"
move.w  #fflag, flg(a0)
trap    #1
; else set flag = F
; and report to system

cm.getd"0"
move.w  dcnt(a0), d0
beq     cm.get0"0"
movea.l  a0, a1
adda.l  #dbeg, a1
movea.l  "4", a2
cm.getd2"0"
move.w  (a1)+, (a2)+
subq.w  #1, d0
bne.s   cm.getd2"0"
; transfer data to data area

cm.get0"0"
clr.b   semi(a0)
; V((a0))

movem.l  (a7)+, "3"
jsr     slice.on
; restore registers

```

```

    endm

; 68881 registers
;
MC68881
    response    equ    0f30000h    ; base address
    cmdnd      equ    MC68881+0
    operand    equ    MC68881+10
; 68881 Macros
; fpcmm: send 68881 command, wait for response
macro fpcmm
    move.w     #1",cmdnd
    cmpi.w     #8900h,response
    beq        wait"0"
endm

wait"0"

; fpwait: wait for data confirmation
macro fpwait
    tst.b     response
    bmi        wait2"0"
endm

wait2"0"

; Other kernel macros
macro getchar
    movem.l   #2",-(a7)
    lea       #1",a0
    jsr       get.c
    movem.l   (a7)+,#2"
endm

macro putchar
    movem.l   #2",-(a7)
    lea       #1",a0
    jsr       put.c
    movem.l   (a7)+,#2"
endm

macro setup.task
    ; save registers
    ; get Q address
    ; get character from Q
    ; retrieve registers

    ; save registers
    ; get Q address
    ; put character in Q
    ; retrieve registers

```

```

move.l a5, cp
movea.l stak(a5), a0
move a0, usp
move.l progr(a5), -(a7)
move.w statr(a5), -(a7)
move.l clock, d0
add.l #slice.time, d0
move.l d0, next.slc(a5)
cmp.w #0, slc.flg(a5)
beq sut.2"0"
c.r.w slc.flg(a5)
move.l ad0(a5), -(a7)
movem.l (a0)+, a1-a6/d0-d7
move.l (a7)+, a0
rte
jsr
rte
endm

sut.2"0"

; set PCB as current process
; get user SP from PCB
; set up as usp
; set up SR, PC for task
; startup
; set up next slice time

; is this a sliced task?
; if so, clear flag
; retrieve registers

macro store.task
move usp, a0
movea.l cp, a5
move.l a0, stak(a5)
move.w (a7)+, statr(a5)
move.l (a7)+, progr(a5)
endm

; save usp
; get current processes PCB
; store SP
; store SR, PC

```

Appendix A2
RTMOS Software Listing

```

;.....
;..          RTMDS
;.....

; board flags (which is this compiled for?)

board4      equ 0
board3      equ 0
board2      equ 1
numboards   equ 3

;.....
;... equates ...
;.....

        if board4
shared      equ board4
myoffs      equ 085000h
        endif
        if board3
shared      equ board3
myoffs      equ 085000h
        endif
        if board2
shared      equ board2
myoffs      equ 5000h
        endif

history     equ shared+2000h
dualport    equ 0f38000h
start_rom   equ 0f80000h ; start of ROM
start_pc    equ start_rom+400h ; start of program
stack       equ 1000h ; starting address of stack

scratchpad  equ 2000h
sur_fail_info equ 0a010h ; starting address of scratchpad ram

; 08081 DUART Registers
;

spbase      equ 0fe8000h ; base address
modea       equ 1 ; mrla,mr2a
sra         equ 3 ; status reg a
csra        equ 3 ; command reg a

```



```

cra      equ      5      ; command reg a
bifa     equ      9      ; aux control reg
acr      equ      9      ; input port config reg
ipcr     equ      0bh    ; int mask reg
imr      equ      0bh    ; int status reg
isr      equ      0bh    ; counter/timer upper reg
ctur     equ      0dh    ; int vector reg
ctlr     equ      0fh    ; output port config reg
ivr      equ      1bh    ; start counter command
opcr     equ      1dh    ; stop counter command
stprt    equ      1fh    ; stop counter command
stprt.loc  equ      spbase+stprt
stprt.loc  equ      spbase+stprt

```

```

;*****
;*** scratchpad variable RAM ***
;*****

```

```

org      scratchpad

clock     long      0
clock.now long      0
valid     word      0
stop.flag word      0
temp      word      0

param1    long      0
param2    long      0
param3    long      0
param4    long      0

```

```

TOPCB.flg word      0
stat.hold word      0
nextsync  long      0

```

```

; local data storage for tasks

```

```

MIXdata   long      0
MIXdata   long      0
MIXdata   long      0
storedata long      0
BIdata    long      0
transdata long      0
Ainvdta   long      0
Cdata     long      0

```

```

m454data      long      0
m453data      long      0
m443data      long      0
det4data      equ       $
               org       $+116
det30data     equ       $
               org       $+52
det31data     equ       $
               org       $+52
det32data     equ       $
               org       $+52
det33data     equ       $
               org       $+52
det34data     equ       $
               org       $+52
det35data     equ       $
               org       $+52
det36data     equ       $
               org       $+52
det37data     equ       $
               org       $+52
det38data     equ       $
               org       $+52
det39data     equ       $
               org       $+52
det310data    equ       $
               org       $+52
det311data    equ       $
               org       $+52
det312data    equ       $
               org       $+52
det313data    equ       $
               org       $+52
det314data    equ       $
               org       $+52
det315data    equ       $
               org       $+52

```

page

```

;*****
;*** Queues
;*****

```

```

ochar      long      obuf,obuf      ; output character queue
obuf      word
          $+254
org
ochar      long      ibuf,ibuf      ; input character queue
ibuf      word
          $+254
org
if board4
word 3,37
syncPCB,outPCB
endif
ready
if board3
word 3,37
syncPCB,outPCB
endif
ready
if board2
word 3,37
syncPCB,outPCB
endif
ready
if board4
word 8,32
m453PCB,de311PCB
endif
timer
if board3
word 9,31
transPCB,m443PCB
endif
timer
if board2
word 9,31
BIPCB,storePCB
endif
timer
cp long 0
Q.done equ $
page
;*****
;see task PCBs
;*****
; PCB offsets

```

```

next      equ 0      ; link to next PCB in Q
stak      equ 4      ; task stack pointer
stakr     equ 8      ; task status register
progr     equ 10     ; current task PC
delta     equ 14     ; timer Q wait time
reiclk    equ 18     ; time to leave timer Q (clock+delta)
TO        equ 22     ; timeout time (on poll)
TOflag    equ 26     ; timeout check in progress flag
slc.flg   equ 28     ; sliced task flag
slc.inh   equ 30     ; slice inhibit flag
nxt.slc   equ 32     ; next slice time
ad0       equ 36     ; special storage for a0 in slice

inPCB     long      innext,inSP      ; character input task
          long      initSR
          long      charin
          long      0,0,0
          word      0
          word      0,1
          word      0,0
          long      3*160
          org
          equ 3      $

inSP
outPCB     long      outnext,outSP      ; character output task
          long      initSR
          long      charout
          long      0,0,0
          word      0
          word      0,1
          word      0,0
          long      3*160
          org
          equ 3      $

outSP
TOPCB      long      0,TOSP
          long      initSR
          long      TO.reporter
          long      0,0,0
          word      0
          word      0,1
          word      0,0
          long      3*160
          org
          equ 3      $

TOSP
syncPCB    long      syncnext,syncSP
          long      initSR
          long      synctask
          word
          long

```

```

syncSP
    long    syncdelta,0,0
    word    0,0,1
    long    0,0
    org     $+160
    equ     $

BIPCB
    long    BInext,BISP
    word    initSR
    long    Bitask
    long    matdelta,waitdelta
    long    0
    word    0,0,1
    long    0,0
    org     $+160
    equ     $

BISP
    long    storenext,storeSP
    word    initSR
    long    storetask
    long    matdelta,waitdelta
    long    0
    word    0,0,1
    long    0,0
    org     $+160
    equ     $

storeSP
    long    KInext,KISP
    word    initSR
    long    Kitask
    long    matdelta,waitdelta
    long    0
    word    0,0,1
    long    0,0
    org     $+160
    equ     $

KISP
    long    transnext,transSP
    word    initSR
    long    transtask
    long    matdelta,waitdelta
    long    0
    word    0,0,1
    long    0,0
    org     $+160
    equ     $

transPCB
    long    transnext,transSP
    word    initSR
    long    transtask
    long    matdelta,waitdelta
    long    0
    word    0,0,1
    long    0,0
    org     $+160
    equ     $

transSP
    long    transnext,transSP
    word    initSR
    long    transtask
    long    matdelta,waitdelta
    long    0
    word    0,0,1
    long    0,0
    org     $+160
    equ     $

```

```

CPCB
    long Cnext,CSP
    initSR
    Ctask
    matdelta,waitdelta
    0
    0,0,1
    0,0
    $+160
    $
    equ

CSP
    long m454next,m454SP
    initSR
    m454task
    matdelta,waitdelta
    0
    0,0,1
    0,0
    $+160
    $
    equ

m454SP
    long m453next,m453SP
    initSR
    m453task
    matdelta,waitdelta
    0
    0,0,1
    0,0
    $+160
    $
    equ

m453SP
    long m443next,m443SP
    initSR
    m443task
    matdelta,waitdelta
    0
    0,0,1
    0,0
    $+160
    $
    equ

m443PCB
    word
    long
    long
    long
    word
    long
    org
    equ

m443SP
    long Ainvnext,AinvSP
    initSR
    Ainvtask
    matdelta,waitdelta
    0
    word
    long
    long
    long
    org
    equ

```

```

AinvSP      word      0,0,1
            long      0,0
            org      $+160
            equ      $
            long      det4next,det4SP
det4PCB      word      initSR
            long      det4task
            long      matdelta,waitdelta
            long      0
            word      0,0,1
            long      0,0
            org      $+160
            equ      $
det4SP      long      det30next,det30SP
            word      initSR
            long      det30task
            long      matdelta,waitdelta
            long      0
            word      0,0,1
            long      0,0
            org      $+160
            equ      $
det30SP      long      det31next,det31SP
            word      initSR
            long      det31task
            long      matdelta,waitdelta
            long      0
            word      0,0,1
            long      0,0
            org      $+160
            equ      $
det31SP      long      det32next,det32SP
            word      initSR
            long      det32task
            long      matdelta,waitdelta
            long      0
            word      0,0,1
            long      0,0
            org      $+160
            equ      $
det32SP      long      det33next,det33SP
            word      initSR
            long      det33task
            long      matdelta,waitdelta
            long      0
            word      0,0,1
            long      0,0
            org      $+160
            equ      $
det33PCB      long      det33next,det33SP

```

```

det33SP
word long
long long
long long
word long
long org
    initSR
    det33task
    matdelta,waitdelta
    0
    0,0,1
    0,0
    3*160
    equ $

det34PCB
long word
word long
long long
long long
word long
long org
    det34next,det34SP
    initSR
    det34task
    matdelta,waitdelta
    0
    0,0,1
    0,0
    3*160
    equ $

det34SP
long word
word long
long long
long long
word long
long org
    det35next,det35SP
    initSR
    det35task
    matdelta,waitdelta
    0
    0,0,1
    0,0
    3*160
    equ $

det35SP
long word
word long
long long
long long
word long
long org
    det36next,det36SP
    initSR
    det36task
    matdelta,waitdelta
    0
    0,0,1
    0,0
    3*160
    equ $

det36PCB
long word
word long
long long
long long
word long
long org
    det37next,det37SP
    initSR
    det37task
    matdelta,waitdelta
    0
    0,0,1
    0,0

```


| | | | |
|-----------|------|----------------------|----|
| det37SP | org | \$+160 equ | \$ |
| det38PCB | long | det38next, det38SP | |
| | word | initSR | |
| | long | det38task | |
| | long | matdelta, waitdelta | |
| | long | 0 | |
| | word | 0, 0, 1 | |
| | long | 0, 0 | |
| det38SP | org | \$+160 equ | \$ |
| det39PCB | long | det39next, det39SP | |
| | word | initSR | |
| | long | det39task | |
| | long | matdelta, waitdelta | |
| | long | 0 | |
| | word | 0, 0, 1 | |
| | long | 0, 0 | |
| det39SP | org | \$+160 equ | \$ |
| det310PCB | long | det310next, det310SP | |
| | word | initSR | |
| | long | det310task | |
| | long | matdelta, waitdelta | |
| | long | 0 | |
| | word | 0, 0, 1 | |
| | long | 0, 0 | |
| det310SP | org | \$+160 equ | \$ |
| det311PCB | long | det311next, det311SP | |
| | word | initSR | |
| | long | det311task | |
| | long | matdelta, waitdelta | |
| | long | 0 | |
| | word | 0, 0, 1 | |
| | long | 0, 0 | |
| det311SP | org | \$+160 equ | \$ |
| det312PCB | long | det312next, det312SP | |
| | word | initSR | |
| | long | det312task | |

```

long      matdelta,waitdelta
long      0
word      0,0,1
long      0,0
org        $+160
equ        $

det312SP
long      det313next,det313SP
word      initSR
long      det313task
long      matdelta,waitdelta
long      0
word      0,0,1
long      0,0
org        $+160
equ        $

det313SP
long      det314next,det314SP
word      initSR
long      det314task
long      matdelta,waitdelta
long      0
word      0,0,1
long      0,0
org        $+160
equ        $

det314SP
long      det315next,det315SP
word      initSR
long      det315task
long      matdelta,waitdelta
long      0
word      0,0,1
long      0,0
org        $+160
equ        $

det315SP
word      tasks.end
equ        $

page

initSR      equ      0200h
; initial Q setup links
if          board4

```


[illegible]

```

matdelta    equ    50          ; Ki done at 20 Hz rate
syncdelta   equ    500         ; system sync every .5 sec
waitdelta   equ    1000        ; initial Ki delay 1 sec

T0.time     equ    00000       ; timeout wait time = 1 min
slice.time  equ    100         ; time till slice = .1 sec

```

page

```

;-----
; Data Structures
; Format: Semaphore
; Producer/Consumer flag
; Data word count
; Data
;-----

```

```

org shared

semi
flag        equ    0
dcnt        equ    2
dbeg        equ    4
cflag       equ    6
pflag       equ    8
fflag       equ    10
; Consumer flag
; Producer flag
; Failure flag

KiStart
word cflag 0,0
word 1
word 0

KiEnd
word cflag 0,0
word 1
word 0

storestart
byte 0,0
word cflag
word 1
word 0

transstart
byte 0,0
word cflag
word 0
word 0,0

transend
byte 0,0

```

| | | | |
|------------|------|-------|-----|
| Cstart | word | cflag | |
| | word | 0 | |
| | | | 0,0 |
| | word | byte | |
| | word | cflag | |
| | | 0 | |
| Cend | word | byte | 0,0 |
| | word | cflag | |
| | word | 0 | |
| m453start | byte | 0,0 | |
| | word | cflag | |
| | word | 0 | |
| m453end | word | byte | 0,0 |
| | word | cflag | |
| | word | 0 | |
| m443start | byte | 0,0 | |
| | word | cflag | |
| | word | 0 | |
| m443end | word | byte | 0,0 |
| | word | cflag | |
| | word | 0 | |
| m454start | byte | 0,0 | |
| | word | cflag | |
| | word | 0 | |
| m454end | word | byte | 0,0 |
| | word | cflag | |
| | word | 0 | |
| Ainvstart | byte | 0,0 | |
| | word | cflag | |
| | word | 0 | |
| Ainvend | word | byte | 0,0 |
| | word | cflag | |
| | word | 0 | |
| det4start | byte | 0,0 | |
| | word | cflag | |
| | word | 32 | |
| | org | 8*64 | |
| det3start | equ | 8 | |
| sblocksize | equ | 42 | |

| | | |
|-------------|------|-------|
| det30start | byte | 0,0 |
| | word | cflag |
| | word | 18 |
| | org | \$+36 |
| det31start | byte | 0,0 |
| | word | cflag |
| | word | 18 |
| | org | \$+36 |
| det32start | byte | 0,0 |
| | word | cflag |
| | word | 18 |
| | org | \$+36 |
| det33start | byte | 0,0 |
| | word | cflag |
| | word | 18 |
| | org | \$+36 |
| det34start | byte | 0,0 |
| | word | cflag |
| | word | 18 |
| | org | \$+36 |
| det35start | byte | 0,0 |
| | word | cflag |
| | word | 18 |
| | org | \$+36 |
| det36start | byte | 0,0 |
| | word | cflag |
| | word | 18 |
| | org | \$+36 |
| det37start | byte | 0,0 |
| | word | cflag |
| | word | 18 |
| | org | \$+36 |
| det38start | byte | 0,0 |
| | word | cflag |
| | word | 18 |
| | org | \$+36 |
| det39start | byte | 0,0 |
| | word | cflag |
| | word | 18 |
| | org | \$+36 |
| det310start | byte | 0,0 |
| | word | cflag |
| | word | 18 |
| | org | \$+36 |
| det311start | byte | 0,0 |
| | word | cflag |

```

word      18
org      $+36
det312start byte 0,0
word      cflag
word      18
org      $+36
det313start byte 0,0
word      cflag
word      18
org      $+36
det314start byte 0,0
word      cflag
word      18
org      $+36
det315start byte 0,0
word      cflag
word      18
org      $+36

det4end   byte 0,0
word      cflag
word      2,0,0

det3end   equ $
eblocksiz equ 10

det30end  byte 0,0
word      cflag
word      2,0,0
det31end  byte 0,0
word      cflag
word      2,0,0
det32end  byte 0,0
word      cflag
word      2,0,0
det33end  byte 0,0
word      cflag
word      2,0,0
det34end  byte 0,0
word      cflag
word      2,0,0
det35end  byte 0,0
word      cflag
word      2,0,0
det36end  byte 0,0
word      cflag

```



```

det37end      word      2,0,0
               byte      0,0
               word      cflag
               word      2,0,0
det38end      byte      0,0
               word      cflag
               word      2,0,0
det39end      byte      0,0
               word      cflag
               word      2,0,0
det310end     word      0,0
               byte      cflag
               word      2,0,0
det311end     byte      0,0
               word      cflag
               word      2,0,0
det312end     byte      0,0
               word      cflag
               word      2,0,0
det313end     byte      0,0
               word      cflag
               word      2,0,0
det314end     byte      0,0
               word      cflag
               word      2,0,0
det315end     byte      0,0
               word      cflag
               word      2,0,0

var.done      equ       8

synctab       word      0,0,0,0

;----- page -----
;see exception vectors see
;-----

resetap      org        start.rom
resetpc      long       stack
              long       start.pc

buserror     long       buserr.handler
addrerror    long       addrerr.handler
illegalinstr long       illinstr.handler
zerodivide   long       zerodiv.handler

```

```

chkinstr      long      chkinstr.handler
trapinstr     long      trapv.handler
priviolation  long      long      privvio.handler
tracevec      long      trace.handler
linel010m    long      linel010.handler
linel111m    long      linel111.handler

long          other.exception
coprocperto  long      coproc.handler
format        long      long      format.handler
uninit        long      long      uninit.handler

org           start.rom+60h
spurious      long      spurious.handler

level1        long      allauto.handler
level2        long      allauto.handler
level3        long      allauto.handler
level4        long      allauto.handler
level5        long      allauto.handler
level6        long      allauto.handler
level7        long      allauto.handler

; kernel calls implemented with 68000 TRAPS

trap0.vec     org       start.rom+80h
trap1.vec     long      rel.routine      ; release
trap2.vec     long      report.sys       ; report-to-system
trap3.vec     long      sleep.routine    ; sleep
trap4.vec     long      rel.on.poll      ; release-on-poll
               long      kill.task       ; terminate

fpcbranch     org       start.rom+60h
fpcpinexact   long      branch.handler
fpcpdivide    long      inexact.handler
fpcpunderfi   long      divide.handler
fpcpoperand   long      underfi.handler
fpcpoverfi    long      operand.handler
fpcpsignal    long      overfi.handler
               long      signal.handler
               long      other.exception
pmmuconfig    long      config.handler
pmmuillegal   long      illegal.handler
pmmuaccess    long      access.handler

org           start.rom+100h

```

```

cntvector    long    tick.handler

;=====
;see
;see initia"tion see
;see
;=====

driver        move.l    #stack,a7                ; set up ssr, sr
               ori.w    #0700h,sr
               jsr      vec.init
               jsr      coninit
               ; move vectors to RAM
               ; init DUART

; clear workspace area

               clr.l    d1
               move.l    #scratchpad,a1
               move.w    d1,(a1)+
               cmpa.l    #0char,a1
               bne.s     dr2

; move initial PCBs, Qs, and data structures to RAM

               jsr      setup.PCBs
               jsr      setup.Qs
               jsr      setup.data

               clr.l    nextsync
               clr.w    synctab+myoffs
               clr.w    TOPCB.fig
               clr.b     surfail

               jsr      cntinit
               lea       ready,a6
               jsr      get.Q
               setup.task

;=====
;see exit trap routine
;=====

rel.routine store.task
               clr.w    T0fig(a5)                ; no longer checking for T0

```

```

; guard against tick
; put this PCB on back of ready
; and get new one from front
; re-enable tick

```

```

ori.w #0700h,sr
lea ready,a6
jsr put.Q
jsr get.Q
andi.w #0f2ffh,sr
setup.task

```

page

```

; Report to system trap
;

```

```

report.sys movem.l a4-a6/d0,-(a7)

```

```

move.w TOPCB.flg,d0
cmp.w #0,d0
bne rep.sys.2

```

```

move.w #1,TOPCB.flg
lea TOPCB,a5
ori.w #0700h,sr
lea ready,a6
jsr put.Q
andi.w #0f2ffh,sr

```

```

rep.sys.2 movem.l (a7)+,a4-a6/d0
rta

```

page

```

; terminate call for one-shot
;
; tasks
;

```

```

kill.task move.w (a7)+,d0
move.l (a7)+,d0
move.l cp,a5
clr.w T0flg(a5)

```

```

ori.w #0700h,sr
lea ready,a6
jsr get.Q
andi.w #0f2ffh,sr
setup.task

```

```

; already a report being
; handled?
; if so then skip
; else, set occupied flag
; set PCB pointer
; guard against tick
; put PCB on back of ready
; re-enable tick

```

```

; remove stacked SR,PC
; no longer checking for T0

```

```

; guard against tick
; get new task from front of ready
; re-enable tick

```

```

;*****
; ** release in poll situation **
; ** same as exit but also set **
; ** up timeout check **
;*****

rel.on.poll store.task

    cmp.w #0,T0flg(a5)
    tne relop2

    move.w #0ffff,T0flg(a5)
    move.l clock,d0
    add.l #T0.time,d0
    move.l d0,T0(a5)

relop2    ori.w #0700h,sr
          lea ready,a5
          jsr put.q
          jsr get.q
          andi.w #0f2ffh,sr
          setup.task

; check.T0 routine used in setup.task macro:

check.T0    move.w T0flg(a5),d0
            cmp.w #0ffff,d0
            bne ch.T0.2

            move.l T0(a5),d0
            cmp.l clock,d0
            bhi ch.T0.2

            move.b #0ffh,d0
            clr.w T0flg(a5)
            rts

ch.T0.2    clr.b d0
            rts
;

page

;*****
; ** release to timer Q trap **
;*****

```

```

; already setup for T0 check?
; if so then skip

```

```

; else, set flag
; setup new timeout (T0) time
; for this PCB

```

```

; guard against tick
; put this PCB on back of ready
; and get new PCB from front
; re-enable tick

```

```

; see if a waiting task

```

```

; get PCB timeout time
; see if past that time

```

```

; if so then flag timeout
; and clear T0 check flag

```

```

; else flag no timeout

```

```

;*****
sleep.routine      store.task

    clr.w    T0flg(a5)
    ori     #0700h,sr
    move.l  clock,d0
    add.l   delta(a5),d0
    move.l  d0,relclk(a5)

    jsr     put.timer
    lea     ready,a6
    jsr     get.Q
    andi.w  #0f2ffh,sr

    setup.task

page

;*****
; ** set up PCBs,Qs,data in RAM **
;*****

; all routines are simply block moves

setup.PCBs  lea    R.inPCB,a0
            lea    inPCB,a1

s.PCB.2     move.b (a0)+,(a1)+
            cmpa.l #tasks.end,a1
            bne    s.PCB.2
            rts

setup.Qs     lea    R.ochar,a0
            lea    ochar,a1

s.Q.2       move.b (a0)+,(a1)+
            cmpa.l #Q.done,a1
            bne    s.Q.2
            rts

setup.data  lea    R.KIstart,a0
            lea    KIstart,a1

s.d.2       move.b (a0)+,(a1)+
            cmpa.l #var.done,a1

```

```

; no longer checking for T0
; guard against tick
; setup time to wake from
; timer Q (clock+delta)

; put PCB into timer Q

; get PCB at front of ready
; re-enable tick

```

```

bne      s.d.2
rts

page

;.....
;ee Q put, get operations  ee
;.....

get.Q    beq      tst.w  (a6)          ; any in Q?
         q.error

         move.l  4(a6),a5             ; get front of Q
         move.l  (a5),4(a6)           ; reset Q front ptr
         sub.w   #1,(a6)              ; update counters
         add.w   #1,2(a6)
         rts

get.Q.2

put.Q    beq      tst.w  2(a6)        ; Q full?
         q.error

         tst.w   (a6)                ; any in Q?
         bne     put.Q.2

         move.l  a5,4(a6)             ; if not
         move.l  a5,8(a6)             ; make front=rear=PCB
         clr.l   (a5)                ; no links
         bra     put.Q.3              ; if so

         move.l  8(a6),a4             ; put PCB at rear of Q
         move.l  a5,(a4)              ; link to current rear
         clr.l   (a5)                ; mark as last (link=nil)
         move.l  a5,8(a6)             ; set rear=PCB

put.Q.3  add.w    #1,2(a6)            ; update counters
         rts

Q.error  jmp      Q.error

put.timer lea     timer,a6
         tst.w   2(a6)                ; Q full?
         beq     Q.error

         tst.w   (a6)                ; any in Q?
         beq     put.Q.4

```

```

put.t.2      move.l 4(a0),a4      ; get front of Q
              moves.l #0,a3
              beq     cmpa.l #0,a4      ; last in Q
              put.t.q.2
              move.l relclk(a4),d1      ; compare clocks
              cmp.l  relclk(a5),d1
              bhi     put.t.3
              moves.l a4,a3
              move.l  (a3),a4
              bra     put.t.2
              ; save this link
              ; goto next in Q

put.t.3      cmpa.l #0,a3      ; put at front of Q?
              beq     put.t.4
              move.l a4,(a5)
              move.l a5,(a3)
              bra     put.q.3
              ; insert into Q

put.t.4      move.l 4(a0),(a5)      ; put at front of Q
              move.l a5,4(a0)
              bra     put.q.3

;=====
; see character Q get/put calls *
;=====
page

get.c        jsr     slice.off
              move.l (a0),d0
              move.l 4(a0),d1
              cmp.l  d0,d1
              beq     get.wait
              moves.l 4(a0),a1
              clr.l  d0
              move.b (a1)+,d0
              moves.l a0,a2
              adda.l #264,a2
              cmpa.l a1,a2
              ; front=rear?
              ; get rear
              ; get char at rear
              ; rear=buf end+1?

```



```

bne      get.2
movea.l  a0,a1          ; if so, set rear=buf start
adda.l   #8,a1

get.2    move.l  a1,4(a0)      ; save rear ptr
         jsr    slice.on
         rts

get.wait move.l  a0,-(a7)      ; wait by releasing CPU
         exit
         move.l  (a7)+,a0
         bra    get.c

put.c    jsr    slice.off
         movea.l (a0),a1
         movea.l 4(a0),a2
         suba.l  #1,a2
         cmpa.l  a1,a2
         beq    put.wait
         suba.l  #7,a2
         cmpa.l  a0,a2
         bne    put.ok
         suba.l  #263,a1
         cmpa.l  a0,a1
         beq    put.wait
         movea.l (a0),a1      ; get front ptr
         move.b  d0,(a1)+      ; put char at front
         movea.l a0,a2
         adda.l  #264,a2
         cmpa.l  a1,a2
         bne    put.2
         movea.l a0,a1
         adda.l  #8,a1          ; if so, set front=buf start

put.2    move.l  a1,(a0)      ; save front ptr
         jsr    slice.on
         rts

put.wait movem.l a0/d0,-(a7)
         exit
         movem.l (a7)+,a0/d0   ; wait by releasing CPU

```

```

bra      put.c

;*****
; ** slice enable, inhibit calls **
;*****

slice.on  move.l  a5,-(a7)          ; get task PCB
          movea.l  cp,a5
          move.w   #0,sic.inh(a5)   ; set flag to enable
          movea.l  (a7)+,a5
          rts

slice.off move.l  a5,-(a7)          ; get task PCB
          movea.l  cp,a5
          move.w   #1,sic.inh(a5)   ; set flag to inhibit
          movea.l  (a7)+,a5
          rts

page

;*****
; ** Task to print system message **
; ** one-shot task put into ready Q by report.sys **
;*****

IO.reporter movea.l #IO.msg,a3      ; output message
          jsr      message
          clr.w    TOPCB.flg       ; let another report
          terminate

;*****
; ** vector transfer to RAM **
;*****

; move vectors from start of ROM to start of RAM
vec.init  movea.l #start.rom,a1
          movea.l #0,a2

vec.2     move.l  (a1)+,(a2)+
          cmpa.l  #start.pc,a1
          bne     vec.2
          rts

;*****
; ** Counter initialization **

```

```

;.....
cntinit      move.l  #0,clock
             lea     spbase,a0
             move.b  #40h,ivr(a0)
             move.b  #00h,imr(a0)
             move.b  #0b0h,scr(a0)
             move.b  #0,ctur(a0)
             move.b  #00h,ctlr(a0)
             andi    #0f2ffh,sr
             tst.b   strt(a0)
             rts

;.....
;... message output routine ...
;...
;... read characters from ...
;... list pointed to by a3, ...
;... output to screen until ...
;... a 0 is found ...
;.....

message      move.b  (a3)+,d0
             cmpi.b  #null,d0
             beq.s   end.msg

             jsr     conout
             bra.s   message

end.msg      rts

;.....
;... valid.hex ...
;...
;... check for valid hex digit ...
;... d0 contains character to be tested ...
;... returns valid set to: ...
;... 0 if invalid hex digit ...
;... 1 if hex number ...
;... 2 if hex letter ...
;.....

valid.hex    clr.b   valid
             cmpi.b  #zero,d0
             bcs.s   end.valid
             cmpi.b  #nine,d0

```

```

        bhi.s    ck.letter
        move.b   #1,valid
        bra.s    end.valid
ck.letter    cmpi.b   #A,d0
             bcs.s   end.valid
             cmpi.b   #F,d0
             bhi.s   end.valid
             move.b   #2,valid
             rts
end.valid

;-----
; Task to sync the clocks of the system
;-----
synctask    move.w   synctab+myoffs,d3      ; update my location
             addi.w   #1,d3
             move.w   d3,synctab+myoffs
             clr.l    d0
             ; set offset to 1st table loc

syncout     move.l   d0,d2
             lsl.l    #1,d2
             lea      synctab,a0
             ; add offset to table base

syncin      move.w   #0(a0,d2),d1
             cmp.w    d3,d1
             bne      syncin
             ; same as mine?
             ; if not then wait
             ; go to next table location

             add.w    #1,d0
             cmp.w    #numboards,d0
             bne      syncout
             ; sync clocks

             move.l   nextsync,d0
             move.l   d0,clock
             add.l     #syncdelta,d0
             move.l   d0,nextsync
             ; update for next sync time

             sleep
             bra      synctask

;-----
; see terminal drivers see
;-----
; acia initialization

```

page

```

coninit      move.l a0,-(a7)
             spbase,a0          ; no parity, 8 bits
             move.b #13h,modea(a0) ; 1 stop
             move.b #37h,modea(a0) ; 9600 baud
             move.b #0bbh,csra(a0) ; disable interrupts
             move.b #0,imr(a0)      ; enable TX/RX
             move.b #5,cra(a0)
             move.l (a7)+,a0
             rts

;*****
;*** console output task ***
;*****

charout      lea    spbase,a0          ; get character from output Q

charo.2      btst  #2,sra(a0)          ; see if ready to transmit
             bne   charo.3
             movem.l d0/a0,-(a7)
             exit
             movem.l (a7)+,d0/a0
             bra   charo.2

charo.3      move.b d0,bufa(a0)        ; send character

check.cr     cmp.b #cr,d0             ; cr?
             bne   end.charout

charo.4      btst  #2,sra(a0)
             beq   charo.4
             move.b #lf,bufa(a0)      ; if so then also send lf

end.charout  bra   charout

page

;*****
;*** console input task ***
;*****

charin       clr.l d0
             lea    spbase,a0

chari.2      btst  #0,sra(a0)          ; see if any received

```

```

bne     chari.3
movem.l d0/a0,-(a7)
exit
movem.l (a7)+,d0/a0
bra     chari.2

chari.3      move.b bufa(a0),d0      ; else get character
             andi.b #7fh,d0         ; strip upper bit
             jsr     upper.lower     ; convert to upper case
             putchar ichar,d0/a0

end.chari    bra     charin

;*****
; character input/output calls **
;*****

conin        getchar ichar,d1-d7/a0-a6
             rts

conout       putchar ochar,d1-d7/a0-a6
             rts

page

;*****
;*** upper.lower ***
;*****

; lower to upper case conversion routine:

upper.lower  cmpi.b #60h,d0
             bis.s  end.up.low
             cmpi.b #7ah,d0
             bhi.s  end.up.low
             subi.b #20h,d0
end.up.low   rts

;*****
;*** bus error ***
;*****

buserr.handler ori.w #0700h,sr
              movea.l a7,a6
              movea.l #2500h,a7

```

```

moves.l #buerr.msg,a3
jsr message

jsr unstack.err
jmp end.exc.routine

;=====
;see unstack.err see
;=====

unstack.err move.b #7,d4
moves.l #display00tab,a3

excloop      jsr message

move.w (a0)+,temp
jsr exc.stack.out

move.b #cr,d0
jsr conout

subi.b #1,d4
bne.s excloop
rts

page

;=====
;see exc.stack.out see
;=====

exc.stack.out move.b temp,d0
jsr two.bytes.out

move.b temp+1,d0
jsr two.bytes.out
rts

;=====
;see other exceptions see
;=====

other.exception moves.l #otherexc.msg,a3
jsr message
jmp end.exc.routine

```

page

```

;=====
;ee Clock tick interrupt handler ee
;=====

tick.handler      tst.b  stop.loc
                  addi.l  #1,clock
                  tst.b  strt.loc

                  move.w  (a7),stat.hold
                  move.l  a1-a6/d0,-(a7)
                  lea     timer,a1
                  tst.w  (a1)
                  beq     slc.check
                        ; any in Q ?

                  movea.l 4(a1),a5
                        ; get first in Q

tick.loop         move.l  relclk(a5),d0
                  cmp.l  clock,d0
                  bne     slc.check
                        ; compare clock values

                  movea.l (a5),a2
                  lea     timer,a6
                  jar     get.q
                  lea     ready,a6
                  jar     put.q
                        ; save link
                        ; release from timer Q
                        ; put on ready Q

                  movea.l a2,a5
                  cmpa.l  #0,a5
                  bne     tick.loop
                        ; last in Q ?
                        ; if not check next in Q

slc.check         andi.w  #2000h,stat.hold ; user task?
                  bne     tick.end

                  movea.l cp,a5
                  move.l  nxt.slc(a5),d0
                  cmp.l  clock,d0
                  bhi     tick.end
                  cmpi.w  #0,slc.inh(a5) ; slice inhibited?
                  bne     tick.end
                  move.w  #1,slc.flg(a5) ; mark as sliced

```



```

move.l a0,ad0(a5)      ; save a0
move    usp,a0
movem.l (a7)+,a1-a6/d0 ; retrieve registers saved above
movem.l a1-a6/d0-d7,-(a0) ; save in task PCB

store.task
ori.w   #0700h,sr
lea     ready,a6
jsr     put.q
jsr     get.q
andi.w  #0f2ffh,sr
setup.task

tick.end  movem.l (a7)+,a1-a6/d0
rte

page

;*****
;*** address error exception ***
;*****
addressr.handler  ori.w   #0700h,sr
movea.l a7,a6
movea.l #2500h,a7
movea.l #addressr.msg,a3
jsr     message
jsr     unstack.err
jmp     end.exc.routine

;*****
;*** illegal instruction exception ***
;*****
illinstr.handler  movea.l #illinstr.msg,a3
jsr     message
jmp     end.exc.routine

;*****
;*** zero divide exception ***
;*****
zerodiv.handler  movea.l #zerodiv.msg,a3
jsr     message
jmp     end.exc.routine

```

```

;*****
;*** CHK instruction exception ***
;*****

chkinstr.handler  movea.l #chkinstr.msg,a3
                  jsr      message
                  jmp      end.exc.routine

                  page

;*****
;*** TRAPV exception ***
;*****

trapv.handler     movea.l #trapv.msg,a3
                  jsr      message
                  jmp      end.exc.routine

;*****
;*** privileged violation exception ***
;*****

privvio.handler   movea.l #privvio.msg,a3
                  jsr      message
                  jmp      end.exc.routine

;*****
;*** trace exception ***
;*****

trace.handler     movea.l #trace.msg,a3
                  jsr      message
                  jmp      end.exc.routine

;*****
;*** line 1010 exception ***
;*****

line1010.handler  movea.l #line1010.msg,a3
                  jsr      message
                  jmp      end.exc.routine

                  page

;*****
;*** line 1111 exception ***
;*****

```

```

linell11.handler  moves.l #linell11.msg,a3
                  jsr      message
                  jmp      end.exc.routine

;=====
;e coprocessor protocol violation exception *
;=====
coproc.handler    moves.l #coproc.msg,a3
                  jsr      message
                  jmp      end.exc.routine

;=====
;e format error exception *
;=====
format.handler    moves.l #format.msg,a3
                  jsr      message
                  jmp      end.exc.routine

;=====
;e uninitialized interrupt exception *
;=====
uninit.handler    moves.l #uninit.msg,a3
                  jsr      message
                  jmp      end.exc.routine

page
;=====
;e spurious interrupt exception **
;=====
spurious.handler  moves.l #spur.msg,a3
                  jsr      message
                  jmp      end.exc.routine

;=====
;e autovector interrupt handler ***
;=====
allauto.handler   moves.l #automesg,a3
                  jsr      message
                  jmp      end.exc.routine

```

```

;.....
;: fpcp branch or set on unordered condition exception *
;.....

branch.handler  movea.l #branch.msg,a3
                jsr    message
                jmp    end.exc.routine

;.....
;: fpcp inexact result exception *
;.....

inexact.handler  movea.l #inexact.msg,a3
                jsr    message
                jmp    end.exc.routine

                page

;.....
;: fpcp divide by zero exception *
;.....

divide.handler  movea.l #divide.msg,a3
                jsr    message
                jmp    end.exc.routine

;.....
;: fpcp underflow exception *
;.....

underfl.handler  movea.l #underfl.msg,a3
                jsr    message
                jmp    end.exc.routine

;.....
;: fpcp operand error exception *
;.....

operand.handler  movea.l #operand.msg,a3
                jsr    message
                jmp    end.exc.routine

;.....
;: fpcp overflow exception *
;.....

overfl.handler  movea.l #overfl.msg,a3

```

```

        jsr    message
        jmp    end.exc.routine

        page

;*****
; fpcp signaling NAN exception *
;*****

signal.handler    move.l    #signal.msg,a3
                  jsr      message
                  jmp      end.exc.routine

;*****
; pmmu configuration exception *
;*****

config.handler    move.l    #config.msg,a3
                  jsr      message
                  jmp      end.exc.routine

;*****
; pmmu illegal operation exception *
;*****

illegal.handler   move.l    #illegal.msg,a3
                  jsr      message
                  jmp      end.exc.routine

;*****
; pmmu access level violation exception *
;*****

access.handler    move.l    #access.msg,a3
                  jsr      message
                  jmp      end.exc.routine

end.exc.routine   move.l    #stack,a7
crash              ori.w    #0700h,sr
                  jmp      crash

        page

;*****
; ASCII Message Strings ***
;*****

```

```

; Timeout reported message
T0.msg      ascii  cr,'Timeout reported to system ',cr,0
; address error message
admesg      ascii  cr,'** Address Error ** ',cr,0
; bus error message
buserr.msg  ascii  cr,'** Bus Error Encountered ** ',cr,0
; current state at error messages
display00tab  ascii  'Instruction Info = ',0
              ascii  'Access Addr High = ',0
              ascii  'Access Addr Low = ',0
              ascii  'Instruction Reg = ',0
              ascii  'Status Reg = ',0
              ascii  'PC High = ',0
              ascii  'PC Low = ',0
; illegal exception error message
otherexc.msg  ascii  cr,'** Illegal Exception Encountered **',cr,0
; address error exception error message
addrerr.msg  ascii  cr,'** Address Error Exception **',cr,0
; illegal instruction message
illinstr.msg  ascii  cr,'** Illegal Instruction Encountered **',cr,0
; division by zero exception message
zerodiv.msg  ascii  cr,'** Zero Divide Exception Encountered **',cr,0
; CHK instruction exception message
chkinstr.msg  ascii  cr,'** CHK Instruction Exception Encountered **',cr,0
; TRAPV exception message
trapv.msg  ascii  cr,'** TRAPV Exception Encountered **',cr,0

```

```

; privileged violation message
privvio.msg      ascii  cr,'** Privileged Violation Encountered **',cr,0
; trace exception message
trace.msg        ascii  cr,'** Trace Exception Encountered **',cr,0
; line1010 exception message
line1010.msg     ascii  cr,'** Line 1010 Exception Encountered **',cr,0
; line1111 exception message
line1111.msg     ascii  cr,'** Line 1111 Exception Encountered **',cr,0
; coprocessor protocol violation exception message
coproc.msg       ascii  cr,'** Coprocessor Protocol Violation Encountered **',cr,0
; format error exception message
format.msg       ascii  cr,'** Format Error Encountered **',cr,0
; uninitialized interrupt exception message
uninit.msg       ascii  cr,'** Uninitialized Interrupt Exception Encountered **',cr,0
; spurious interrupt exception message
spur.msg         ascii  cr,'** Spurious Interrupt Encountered **',cr,0
; autovector interrupt message
automesg         ascii  cr,'** Autovector Interrupt Encountered **',cr,0
; FPCP branch or set on unordered condition exception message
branch.msg       ascii  cr,'** FPCP Branch or Set on Unordered Condition Encountered *
; FPCP inexact result exception message
inexact.msg      ascii  cr,'** FPCP Inexact Result Exception Encountered **',cr,0
; FPCP divide by zero exception message

```

```

divide.mesg ascii cr,'** FPCP Divide by Zero Exception Encountered **',cr,0
; FPCP underflow exception message
underfl.mesg      ascii cr,'** FPCP Underflow Exception Encountered **',cr,0
; FPCP operand error exception message
operand.mesg      ascii cr,'** FPCP Operand Error Encountered **',cr,0
; FPCP overflow exception message
overfl.mesg ascii cr,'** FPCP Overflow Exception Encountered **',cr,0
; FPCP signaling NAN exception message
signal.mesg ascii cr,'** FPCP Signaling NAN Exception Encountered **',cr,0
; PMMU configuration exception message
config.mesg ascii cr,'** PMMU Configuration Exception Encountered **',cr,0
; PMMU illegal operation exception message
illegal.mesg      ascii cr,'** PMMU Illegal Operation Exception Encountered **',cr,0
; PMMU access level violation exception message
access.mesg ascii cr,'** PMMU Access Level Violation Encountered **',cr,0

;*****
; ROM init copy of PCBs, Qs, and data *
; for info see RAM copy above *
;*****
R.ochar      word      long      obuf,obuf
org          $+254
R.ichar      word      long      ibuf,ibuf
org          $+254
R.ready      if        board4    3,37

```



```

R.ready      syncPCB,outPCB
long
endif
if
board3
word 3,37
syncPCB,outPCB
long
endif
if
board2
word 3,37
syncPCB,outPCB
long
endif

R.timer      board4
word 8,32
m453PCB,del311PCB
long
endif
if
board3
word 9,31
transPCB,m443PCB
long
endif
if
board2
word 9,31
BIPCB,storePCB
long
endif

R.cp         long 0

R.inPCB      long innext,inSP
word
initSR
charin
0,0,0
word 0
word 0,1
word 0,0
org $+160

R.outPCB     long outnext,outSP
word
initSR
charout
0,0,0
word 0
word 0,1
word 0,0
org $+160

R.TOPCB      long 0,TOSP
word
initSR

```

; ready Q

; ready Q

; timer Q

; timer Q

| | | |
|------------|------|---------------------|
| | | T0.reporter |
| | | 0,0,0 |
| | | 0 |
| | | 0,1 |
| | | 0,0 |
| | | \$+160 |
| R.syncPCB | long | syncnext, syncSP |
| | word | initSR |
| | long | synctask |
| | long | syncdelta, 0, 0 |
| | word | 0, 0, 1 |
| | long | 0, 0 |
| | org | \$+160 |
| R.BIPCB | | long BInext, BISP |
| | word | initSR |
| | long | Bitask |
| | long | matdelta, waitdelta |
| | long | 0 |
| | word | 0, 0, 1 |
| | long | 0, 0 |
| | org | \$+160 |
| R.storePCB | long | storenext, storeSP |
| | word | initSR |
| | long | storetask |
| | long | matdelta, waitdelta |
| | long | 0 |
| | word | 0, 0, 1 |
| | long | 0, 0 |
| | org | \$+160 |
| R.KIPCB | | long KInext, KISP |
| | word | initSR |
| | long | Kitask |
| | long | matdelta, waitdelta |
| | long | 0 |
| | word | 0, 0, 1 |
| | long | 0, 0 |
| | org | \$+160 |
| R.transPCB | long | transnext, transSP |
| | word | initSR |
| | long | transtask |
| | long | matdelta, waitdelta |

| | | | |
|-----------|-------------------------------------------------------------|--------------------------------------------------------------------------------------------|-----------|
| R.CPCB | long word long org | 0 0,0,1 0,0 \$+160 | |
| | word long long long word long org | long initSR Ctask matdelta,waitdelta 0 0,0,1 0,0 \$+160 | Cnext,CSP |
| R.m454PCB | long word long long long word long org | m454next,m454SP initSR m454task matdelta,waitdelta 0 0,0,1 0,0 \$+160 | |
| R.m453PCB | long word long long long word long org | m453next,m453SP initSR m453task matdelta,waitdelta 0 0,0,1 0,0 \$+160 | |
| R.m443FCR | long word long long long word long org | m443next,m443SP initSR m443task matdelta,waitdelta 0 0,0,1 0,0 \$+160 | |
| R.invPCB | long word long long long | Ainvnext,AinvSP initSR Ainvtask matdelta,waitdelta 0 | |

| | | | |
|------------|------|---------------------|--|
| R.det4PCB | word | 0,0,1 | |
| | long | 0,0 | |
| | org | \$+160 | |
| | long | det4next, det4SP | |
| | word | initSR | |
| | long | det4task | |
| | long | matdelta, waitdelta | |
| | long | 0 | |
| | word | 0,0,1 | |
| | long | 0,0 | |
| | org | \$+160 | |
| R.det30PCB | long | det30next, det30SP | |
| | word | initSR | |
| | long | det30task | |
| | long | matdelta, waitdelta | |
| | long | 0 | |
| | word | 0,0,1 | |
| | long | 0,0 | |
| | org | \$+160 | |
| R.det31PCB | long | det31next, det31SP | |
| | word | initSR | |
| | long | det31task | |
| | long | matdelta, waitdelta | |
| | long | 0 | |
| | word | 0,0,1 | |
| | long | 0,0 | |
| | org | \$+160 | |
| R.det32PCB | long | det32next, det32SP | |
| | word | initSR | |
| | long | det32task | |
| | long | matdelta, waitdelta | |
| | long | 0 | |
| | word | 0,0,1 | |
| | long | 0,0 | |
| | org | \$+160 | |
| R.det33PCB | long | det33next, det33SP | |
| | word | initSR | |
| | long | det33task | |
| | long | matdelta, waitdelta | |
| | long | 0 | |
| | word | 0,0,1 | |

```

long      0,0
org      $+160

R.det34PCB  long      det34next,det34SP
            word      initSR
            long      det34task
            long      matdelta,waitdelta
            long      0
            word      0,0,1
            long      0,0
            org      $+160

R.det35PCB  long      det35next,det35SP
            word      initSR
            long      det35task
            long      matdelta,waitdelta
            long      0
            word      0,0,1
            long      0,0
            org      $+160

R.det36PCB  long      det36next,det36SP
            word      initSR
            long      det36task
            long      matdelta,waitdelta
            long      0
            word      0,0,1
            long      0,0
            org      $+160

R.det37PCB  long      det37next,det37SP
            word      initSR
            long      det37task
            long      matdelta,waitdelta
            long      0
            word      0,0,1
            long      0,0
            org      $+160

R.det38PCB  long      det38next,det38SP
            word      initSR
            long      det38task
            long      matdelta,waitdelta
            long      0
            word      0,0,1
            long      0,0

```

```

org      $+160
R.det39PCB  long  det39next,det39SP
           word  initSR
           long  det39task
           long  matdelta,waitdelta
           long  0
           word  0,0,1
           long  0,0
           org   $+160

R.det310PCB long  det310next,det310SP
           word  initSR
           long  det310task
           long  matdelta,waitdelta
           long  0
           word  0,0,1
           long  0,0
           org   $+160

R.det311PCB long  det311next,det311SP
           word  initSR
           long  det311task
           long  matdelta,waitdelta
           long  0
           word  0,0,1
           long  0,0
           org   $+160

R.det312PCB long  det312next,det312SP
           word  initSR
           long  det312task
           long  matdelta,waitdelta
           long  0
           word  0,0,1
           long  0,0
           org   $+160

R.det313PCB long  det313next,det313SP
           word  initSR
           long  det313task
           long  matdelta,waitdelta
           long  0
           word  0,0,1
           long  0,0
           org   $+160

```

| | | |
|--------------|------|----------------------|
| R.det314PCB | long | det314next, det314SP |
| | word | initSR |
| | long | det314task |
| | long | matdelta, waitdelta |
| | long | 0 |
| | word | 0, 0, 1 |
| | long | 0, 0 |
| | org | \$+160 |
| R.det315PCB | long | det315next, det315SP |
| | word | initSR |
| | long | det315task |
| | long | matdelta, waitdelta |
| | long | 0 |
| | word | 0, 0, 1 |
| | long | 0, 0 |
| | org | \$+160 |
| R.KIstart | byte | 0, 0 |
| | word | cflag |
| | word | 1 |
| | word | 0 |
| R.KIend | word | byte 0, 0 |
| | word | cflag |
| | word | 1 |
| | word | 0 |
| R.storestart | byte | 0, 0 |
| | word | cflag |
| | word | 1 |
| | word | 0 |
| R.transstart | byte | 0, 0 |
| | word | cflag |
| | word | 0 |
| R.transend | byte | 0, 0 |
| | word | cflag |
| | word | 0 |
| R.Cstart | byte | 0, 0 |
| | word | cflag |
| | word | 0 |
| R.Cend | word | byte 0, 0 |
| | word | cflag |
| | word | 0 |

```

R.m453start  byte 0,0
               word cflag
               word 0
R.m453end    byte 0,0
               word cflag
               word 0
R.m443start  byte 0,0
               word cflag
               word 0
R.m443end    byte 0,0
               word cflag
               word 0
R.m454start  byte 0,0
               word cflag
               word 0
R.m454end    byte 0,0
               word cflag
               word 0
R.Ainvstart  byte 0,0
               word cflag
               word 0
R.Ainvend    byte 0,0
               word cflag
               word 0
R.det4start  byte 0,0
               word cflag
               word 32
               org $+64
R.det3start  equ $
R.det30start byte 0,0
               word cflag
               word 18
               org $+36
R.det31start byte 0,0
               word cflag
               word 18
               org $+36
R.det32start byte 0,0
               word cflag

```



```

word 18
org $+36
R.det33start byte 0,0
word cflag
word 18
org $+36
R.det34start byte 0,0
word cflag
word 18
org $+36
R.det35start byte 0,0
word cflag
word 18
org $+36
R.det36start byte 0,0
word cflag
word 18
org $+36
R.det37start byte 0,0
word cflag
word 18
org $+36
R.det38start byte 0,0
word cflag
word 18
org $+36
R.det39start byte 0,0
word cflag
word 18
org $+36
R.det310start byte 0,0
word cflag
word 18
org $+36
R.det311start byte 0,0
word cflag
word 18
org $+36
R.det312start byte 0,0
word cflag
word 18
org $+36
R.det313start byte 0,0
word cflag
word 18
org $+36

```

| | | | |
|---------------|------|-------|-----|
| R.det314start | word | byte | 0,0 |
| | word | cflag | |
| | org | 18 | |
| R.det315start | org | \$+36 | 0,0 |
| | word | byte | |
| | word | cflag | |
| | org | 18 | |
| | org | \$+36 | |
| R.det4end | byte | 0,0 | |
| | word | cflag | |
| | word | 2,0,0 | |
| R.det3end | equ | \$ | |
| R.det30end | byte | 0,0 | |
| | word | cflag | |
| | word | 2,0,0 | |
| R.det31end | byte | 0,0 | |
| | word | cflag | |
| | word | 2,0,0 | |
| R.det32end | byte | 0,0 | |
| | word | cflag | |
| | word | 2,0,0 | |
| R.det33end | byte | 0,0 | |
| | word | cflag | |
| | word | 2,0,0 | |
| R.det34end | byte | 0,0 | |
| | word | cflag | |
| | word | 2,0,0 | |
| R.det35end | byte | 0,0 | |
| | word | cflag | |
| | word | 2,0,0 | |
| R.det36end | byte | 0,0 | |
| | word | cflag | |
| | word | 2,0,0 | |
| R.det37end | byte | 0,0 | |
| | word | cflag | |
| | word | 2,0,0 | |
| R.det38end | byte | 0,0 | |
| | word | cflag | |
| | word | 2,0,0 | |
| R.det39end | byte | 0,0 | |
| | word | cflag | |
| | word | 2,0,0 | |
| R.det310end | byte | 0,0 | |

| | | |
|-------------|------|-------|
| R.det311end | word | cflag |
| | word | 2,0,0 |
| | byte | 0,0 |
| | word | cflag |
| | word | 2,0,0 |
| R.det312end | word | cflag |
| | byte | 0,0 |
| | word | 2,0,0 |
| | word | cflag |
| | word | 2,0,0 |
| R.det313end | word | cflag |
| | byte | 0,0 |
| | word | 2,0,0 |
| | word | cflag |
| | word | 2,0,0 |
| R.det314end | word | cflag |
| | byte | 0,0 |
| | word | 2,0,0 |
| | word | cflag |
| | word | 2,0,0 |
| R.det315end | word | cflag |
| | byte | 0,0 |
| | word | 2,0,0 |
| | word | cflag |
| | word | 2,0,0 |

Appendix A3
Applications Tasks Listing

```

;*****
;***** Applications tasks *****
;*****
;***** 8081 interface routines *****
;*****
; BItask: form BI from B0 by copying all but column of failed surface
;*****

BItask      move.b    surfail,d0      ; get failure data word
            beq      LFfail          ; if no fail then signal as
                                      ; such

            btst     #0,d0           ; left elevator failure?
            beq      LEfail
            bra      setBI

            btst     #1,d0           ; right elevator failure?
            beq      REfail
            bra      setBI

            btst     #2,d0           ; left aileron failure?
            beq      LAfail
            bra      setBI

            btst     #3,d0           ; right aileron failure?
            beq      RAfail
            bra      setBI

            btst     #4h,d0          ; no flaps in model so act
            beq      LFfail          ; as if no fail
            bra      setBI

            btst     #5h,d0
            beq      RFfail
            bra      setBI

            btst     #6h,d0          ; rudder failure?
            beq      BI7
            bra      setBI

```

```

Rudfail    beq    LFfail
            move.b #4,d1
setBI      move.b d1,BIdata
            ; prepare to set up BI
            ; set pointers for transfer

            lea    B0,a0
            lea    BI,a1
            clr.l  d2

            ; previous routine signals
            ; column to skip

setBI6     cmp.b  d1,d2
            bne    setBI4
            adda.l #20,a0
            bra    setBI5
            ; this column?
            ; if so then skip

setBI4     clr.l  d3

setBI3     move.l (a0)+,(a1)+
            add.b  #1,d3
            cmp.b  #5,d3
            bne    setBI3
            ; else copy to BI

setBI5     add.b  #1,d2
            cmp.b  #5,d2
            bne    setBI6
            ; done all columns?

            p.poll KIstart,BIdata
            c.poll KIend,d0-d1,BIdata
            sleep
            ; start KI computations
            ; wait for results

            bra    BITask

; storetask: take KI matrix computed, convert to scaled integer,
;             and store in dual port for 8061

storetask  c.poll storestart,d0-d1,storedata
            move.b storedata,d0
            cmp.b  #0fffh,d0
            beq    storenone
            lea    KI,a0
            ; set source, destination pointers

```

```

lea      MIX,a1
clr.l    d2

stt4     clr.l    d3
stt5     cmp.b    d3,d0          ; failed surface?
bne      stt9
stt8     move.l    #0,d5        ; if so store 0
jsr      storeMIX
bra      stt3
stt9     cmp.b    #4,d3          ; check for flaps (not used)
beq      stt8
cmp.b    #5,d3
beq      stt8

stt2     move.l    (a0)+,d5      ; else get data from KI
jsr      convMIX                ; convert to scaled integer
jsr      storeMIX              ; store in dual port

stt3     add.b    #1,d3          ; done with this column?
cmp.b    #7,d3
bne      stt5
add.b    #1,d2
cmp.b    #3,d2
bne      stt4
; done with all rows?

sleep
bra      storetask

storenone lea      MIX,a1          ; if no fail, then store
lea      nofail,a0              ; nofail matrix
clr.l    d2

stt7     move.w    (a0)+,(a1)+
add.b    #1,d2
cmp.b    #21,d2
bne      stt7
sleep
bra      storetask

; convMIX: convert from floating point to scaled integer
; (4 bits to right of decimal point)

```

```

convMIX      move.l #41800000h,d6 ; multiply by 16
             jsr      mult65
             jsr      int5
             rts      ; take integer

int5         fpcomm #4400h
             move.l  d5,operand
             fwait
             fpcomm #6000h
             move.l  operand,d5
             fwait
             rts

storeMIX     clr.l  d4
             move.b d3,d4
             mulu   #6,d4
             move.l d2,d6
             lsl.l  #1,d6
             add.w  d6,d4

             move.w d5,0(a1,d4) ; store in destination matrix
             rts

             page

;*****
; KI = inv(BIT * BI) * BIT * BOK0 computation routines **
;*****
KItask       move.l clock,history
             c.poll  K1start,d0-d1,KIdata ; wait for signal to start

             move.l clock,history+4
             p.poll  transstart,KIdata
             c.poll  transend,d0-d1,KIdata

             move.l clock,history+8
             p.poll  Cstart,KIdata

             p.poll  m453start,KIdata
             c.poll  m453end,d0-d1,KIdata

             move.l clock,history+12

```



```

c.poll Cend,d0-d1,KIdata
move.l clock,history+16
p.poll m443start,KIdata
c.poll m443end,d0-d1,KIdata

move.l clock,history+20
p.poll storestart,KIdata
p.poll Klend,KIdata

;sleep
bra KItask

transtask
move.l clock,history+24
c.poll transstart,d0-d1,transdata
move.l clock,history+28
jsr transBI
move.l clock,history+32
p.poll transend,transdata

sleep
bra transtask

Ctask
move.l clock,history+36
c.poll Cstart,d0-d1,Cdata

move.l clock,history+40
p.poll m454start,Cdata
c.poll m454end,d0-d1,Cdata

move.l clock,history+44
p.poll Ainvsstart,Cdata
c.poll Ainvend,d0-d1,Cdata

move.l clock,history+48
p.poll Cend,Cdata

sleep
bra Ctask

m454task
move.l clock,history+52
c.poll m454start,d0-d1,m454data

move.l clock,history+56
move.l #4,d0
move.l #5,d1

```

```

move.l #4,d2
lea BIT,a1
lea BI,a2
lea AMX,a3
jsr mmul

move.l clock,history+60
p.poll m454end,m454data

sleep
bra m454task

m453task
move.l clock,history+64
c.poll m453start,d0-d1,m453data

move.l clock,history+68
move.l #4,d0
move.l #5,d1
move.l #3,d2
lea BIT,a1
lea BOKO,a2
lea DMX,a3
jsr mmul

move.l clock,history+72
p.poll m453end,m453data

sleep
bra m453task

m443task
move.l clock,history+76
c.poll m443start,d0-d1,m443data

move.l clock,history+80
move.l #4,d0
move.l #4,d1
move.l #3,d2
lea CMX,a1
lea DMX,a2
lea KI,a3
jsr mmul

move.l clock,history+84
p.poll m443end,m443data

sleep

```

```

det4task
    bra      m443task
    move.l   clock,history+88
    c.poll   det4start,d0-d1,det4data

    move.l   clock,history+92
    lea      det4data,a1
    move.l   #4,d4
    jsr      deter
    move.l   d5,det4data

    move.l   clock,history+96
    p.poll   det4end,det4data

    sleep
    bra      det4task

det30task
    c.poll   det30start,d0-d1,det30data

    lea      det30data,a1
    suba.l   #64,a1
    move.l   #3,d4
    jsr      deter
    move.l   d5,det30data

    p.poll   det30end,det30data

    sleep
    bra      det30task

det31task
    c.poll   det31start,d0-d1,det31data

    lea      det31data,a1
    suba.l   #64,a1
    move.l   #3,d4
    jsr      deter
    move.l   d5,det31data

    p.poll   det31end,det31data

    sleep
    bra      det31task

det32task
    c.poll   det32start,d0-d1,det32data

    lea      det32data,a1

```

```

suba.l #64,a1
move.l #3,d4
jsr    deter
move.l d5,det32data

p.poll det32end,det32data

sleep
bra    det32task

det33task c.poll det33start,d0-d1,det33data

lea     det33data,a1
suba.l #64,a1
move.l #3,d4
jsr    deter
move.l d5,det33data

p.poll det33end,det33data

sleep
bra    det33task

det34task c.poll det34start,d0-d1,det34data

lea     det34data,a1
suba.l #64,a1
move.l #3,d4
jsr    deter
move.l d5,det34data

p.poll det34end,det34data

sleep
bra    det34task

det35task c.poll det35start,d0-d1,det35data

lea     det35data,a1
suba.l #64,a1
move.l #3,d4
jsr    deter
move.l d5,det35data

p.poll det35end,det35data

```

```

det36task      sleep      det35task
                bra
                c.poll    det36start,d0-d1,det36data
                lea      det36data,a1
                suba.l   #64,a1
                move.l   #3,d4
                jsr      deter
                move.l   d5,det36data
                p.poll    det36end,det36data

                sleep
                bra      det36task

det37task      c.poll    det37start,d0-d1,det37data
                lea      det37data,a1
                suba.l   #64,a1
                move.l   #3,d4
                jsr      deter
                move.l   d5,det37data
                p.poll    det37end,det37data

                sleep
                bra      det37task

det38task      c.poll    det38start,d0-d1,det38data
                lea      det38data,a1
                suba.l   #64,a1
                move.l   #3,d4
                jsr      deter
                move.l   d5,det38data
                p.poll    det38end,det38data

                sleep
                bra      det38task

det39task      c.poll    det39start,d0-d1,det39data
                lea      det39data,a1
                suba.l   #64,a1

```

```

move.l #3,d4
jsr    deter
move.l d5,det39data

p.poll det39end,det39data

sleep
bra    det39task

det310task c.poll det310start,d0-d1,det310data

lea     det310data,a1
suba.l  #64,a1
move.l  #3,d4
jsr     deter
move.l  d5,det310data

p.poll  det310end,det310data

sleep
bra     det310task

det311task c.poll det311start,d0-d1,det311data

lea     det311data,a1
suba.l  #64,a1
move.l  #3,d4
jsr     deter
move.l  d5,det311data

p.poll  det311end,det311data

sleep
bra     det311task

det312task c.poll det312start,d0-d1,det312data

lea     det312data,a1
suba.l  #64,a1
move.l  #3,d4
jsr     deter
move.l  d5,det312data

p.poll  det312end,det312data

sleep

```

```

bra    det312task

det313task  c.poll  det313start,d0-d1,det313data
            lea     det313data,a1
            suba.l  #64,a1
            move.l  #3,d4
            jsr     deter
            move.l  d5,det313data
            p.poll  det313end,det313data
            sleep
            bra     det313task

det314task  c.poll  det314start,d0-d1,det314data
            lea     det314data,a1
            suba.l  #64,a1
            move.l  #3,d4
            jsr     deter
            move.l  d5,det314data
            p.poll  det314end,det314data
            sleep
            bra     det314task

det315task  c.poll  det315start,d0-d1,det315data
            lea     det315data,a1
            suba.l  #64,a1
            move.l  #3,d4
            jsr     deter
            move.l  d5,det315data
            p.poll  det315end,det315data
            sleep
            bra     det315task

; matrix multiply
mmul1      clr.l   d3          ; for i = 0 to m-1
mmult2     clr.l   d4          ; for j = 0 to p-1

```

```

mmult3      clr.l d8      d5      ; accum=0
            ; for k = 0 to n-1

mmult4      jsr      matern      ; multiply and accumulate
            ; next k

            jsr      storeCij
            ; next j

            add.w #1,d4
            cmp.w d4,d2
            bne      mmult3

            add.w #1,d3
            cmp.w d3,d0
            bne      mmult2

            rts

matern      movem.l d0-d3/d6,-(a7)
            movea.l a1,a4
            movea.l a2,a5
            lsl.l #2,d0
            lsl.l #2,d1
            ; Acoloff
            ; Bcoloff

            move.l d6,d2
            mulu d2,d0
            move.l d4,d2
            mulu d2,d1
            ; Acoloff * k
            ; Bcoloff * j

            lsl.l #2,d3
            adda.l d3,a4
            lsl.l #2,d6
            adda.l d6,a5
            ; Arowoff
            ; Browoff

            move.l 0(a4,d0),d2
            move.l 0(a5,d1),d1
            ; A(i,k)
            ; B(k,j)

            jsr      mult12
            ; A(i,k) * B(k,j)

            jsr      add25
            ; add to accum

            movem.l (a7)+,d0-d3/d6

```



```

storeCij
    rts
    movem.l d0-d3,-(a7)
    movea.l a3,a4
    lsl.l #2,d0
    move.l d4,d1
    mulu d1,d0
    lsl.l #2,d3
    adda.l d3,a4
    move.l d5,0(a4,d0)
    movem.l (a7)+,d0-d3
    rts

mult12
    fpcmm #4400h
    move.l d1,operand
    fpwait
    fpcmm #4423h
    move.l d2,operand
    fpwait
    fpcmm #6400h
    move.l operand,d2
    fpwait
    rts

add25
    fpcmm #4400h
    move.l d2,operand
    fpwait
    fpcmm #4422h
    move.l d5,operand
    fpwait
    fpcmm #6400h
    move.l operand,d5
    fpwait
    rts

page

Acoloff equ 16
Bcoloff equ 12
Ccoloff equ 16
; Ccoloff
; Ccoloff * j
; Crowoff
; store at C(i,j)

```

```

macro      getm
clr.l     d1
move.w    #4",d1
mulu      #2",d1
movea.l   #1",a0
adda.w    d1,a0
move.w    #3",d1
lsl.w     #2,d1
move.l    0(a0,d1),d0
endm

macro      putm
clr.l     d1
move.w    #4",d1
mulu      #2",d1
movea.l   #1",a0
adda.w    d1,a0
move.w    #3",d1
lsl.w     #2,d1
move.l    d0,0(a0,d1)
endm

```

```

; matrix inverse

```

```

Ainvtask   c.poll  Ainvstart,d0-d1,Ainvdata

```

```

p.poll     det4start,AMX

```

```

lea        AMX,a1
lea        CMX,a2
movea.l    a1,a3
adda.l     #64,a3

```

```

cofactor   clr.l   d2                      ; for i=0 to n

```

```

cofloopp1  clr.l   d3                      ; for j=0 to n

```

```

cofloopp2  clr.l   d4                      ; y=0
clr.l      d5                          ; y2=0

```

```

cof6        cmp.w   d3,d4                  ; if y<>j then

```

```

beq    cof5
clr.l  d6          ; x=0
clr.l  d7          ; x2=0

cof4   cmp.w  d6,d2      ; if x<>i then
      cof3

      getm    a1,Acoloff,d6,d4 ; copy A(x,y) to B(x2,y2)
      putm    a3,Bcoloff,d7,d5
      zdd.w   #1,d7      ; x2=x2+1

cof3   add.w   #1,d6      ; x=x+1
      cmp.w   #4,d6      ; if x<>n then loop back
      bne     cof4

      add.w   #1,d5      ; y2=y2+1

cof5   add.w   #1,d4      ; y=y+1
      cmp.w   #4,d4      ; if y<>n then loop back
      bne     cof6

      movem.l a0-a2/d0,-(a7)
      move.l  d3,d5
      lsl.l   #2,d5
      add.l   d2,d5
      mulu    #sblocksiz,d5
      p.poll  det3start,d5,a3
      movem.l (a7)+,a0-a2/d0

      add.w   #1,d3      ; next j
      cmp.w   #4,d3
      bne     cofloop2

      add.w   #1,d2      ; next i
      cmp.w   #4,d2
      bne     cofloop1

      c.poll  det4end,a0-a3,Ainvdata
      move.l  Ainvdata,-(a7)

Ain2   clr.l   d2          ; for i = 0 to n
Ain3   clr.l   d3          ; for j = 0 to n
Ain4   move.l  d3,d5

```

```

        lsl.l  #2,d5
        add.l  d2,d5
        mulu   #eblocksize,d5
        c.pollm det3end,d5,a0-a2/d0-d3,#Ainvdata ; get det(A(i,j))
        move.l Ainvdata,d7

        move.w d2,d4
        add.w  d3,d4
        and.w  #1,d4
        beq    Ain5
        jsr    neg7

        ; determine sign = (-1)**(i+j)

Ain5:   move.l  d7,d0
        move.l  (a7),d7
        jsr     div70
        putm    a2,Ccoloff,d3,d2 ; divide by det(A
                                   ; store at C(j,i)

        add.w   #1,d3
        cmp.w   #4,d3
        bne     Ain4
        ; next j

        add.w   #1,d2
        cmp.w   #4,d2
        bne     Ain3
        ; next i

        move.l  (a7)+,d7
        p.poll  Ainvend,Ainvdata
        sleep   bra    Ainvtask
        page

        ; matrix determinate routine
deter   movem.l d2-d4/d6-d7/a6,-(a7)
        cmp.w   #2,d4
        beq     det3
        ; simple determinate? (n=2)

        clr.l   d2
        clr.l   d3
        ; if not: accum=0
        ; for k = 0 to n-1

        move.l  #3f800000h,d6 ; sign of a(k,0) term
        jsr     copymat ; copy A(k,0) to next lower order
        det2

```

```

sub.w    #1,d4
jsr      deter
add.w    #1,d4

jsr      getan
jsr      mult05
jsr      mult05
jsr      neg6
jsr      add52

add.w    #1,d3
cmp.w    d3,d4
bne      det2

move.l   d2,d5
movem.l  (a7)+,d2-d4/d6-d7/a6
rts

det3
movem.l  (a7)+,d2-d4/d6-d7/a6
jsr      simple
rts

page

copymat
copy2
movem.l  d0-d4/a4-a5,-(a7)
cmp.w    #4,d4
bne      copy3

movea.l  a1,a4
adda.l   #16,a4
movea.l  a1,a5
adda.l   #64,a5
movea.l  a1,a6
bra      copy4

copy3
movea.l  a1,a4
adda.l   #76,a4
movea.l  a1,a5
adda.l   #100,a5
movea.l  a1,a6
adda.l   #64,a6

copy4
move.w   #1,d1
clr.l    d0

copy5
; matrix, return a3=matrix pointer
; take determinate of minor matrix
; will leave out row=k, col=0

; get a(k,0)
; take doter(A(k,0))*a(k,0)
; establish sign of term
; switch sign for next time
; add to accum

; next k

; A4 + 16 (2nd col)
; A3
; A4
; A3 + 12 (2nd col)
; for col = 1 to n-1
; for row = 0 to n-1

```

```

copy7      cmp.w  d0,d3      ; skip row = k
           beq    copy6

           move.l (a4),(a5)+  ; copy matrix entry

copy6      adda.l #4,a4
           add.w  #1,d0      ; next row
           cmp.w  d0,d4
           bne   copy7

           add.w  #1,d1      ; next col
           cmp.w  d1,d4
           bne   copy5

           movem.l (a7)+,d0-d4/a4-a5
           rts

page

getan      move.l d3,d0      ; row #
           lsl.l #2,d0      ; long word
           move.l 0(a6,d0),d0 ; get a(k,0)
           rts

simple      movea.l a1,a0     ; get A2 address
           adda.l #100,a0

           move.l (a0),d0
           move.l 12(a0),d5
           jsr   mult05      ; a11 * a22
           move.l d5,d2

           move.l 4(a0),d0
           move.l 8(a0),d5
           jsr   mult05      ; a12 * a21

           jsr   sub52      ; a11*a22 - a12*a21
           move.l d2,d5      ; return d5=determinate
           rts

page

matdata   equ    shared+1000h
BIT       equ    matdata
AMX       equ    matdata+100h

```

```

CMX      equ      matdata+200h
DMX      equ      matdata+300h
KI       equ      matdata+400h
BI       equ      matdata+500h
MIX      equ      dualport+20h
surfail  equ      dualport+11h

; control mixer data
B0       long     0bb5ed289h,0bf14c986h,0,3e7b645ah,0bc54fdf4h
         long     0bb5ed289h,0bf14c986h,0,0be7b645ah,3c54fdf4h
         long     0bb102de0h,0bd450481h,0,3f2b7176h,0bd2f4f0eh
         long     3b102de0h,3d450481h,0,3f2b7176h,0bd2f4f0eh
         long     0,0,3ad1b717h,3d62eb1ch,0be373190h

B0K0     long     0bbdded289h,0bf94c986h,0,0,0
         long     0,0,0,3fab7176h,0bdaf4f0eh
         long     0,0,3ad1b717h,3d62eb1ch,0be373190h

nofail   word     0010h,0,0,0010h,0,0,0,0010h,0
         word     0,0010h,0,0,0,0,0,0,0
         word     0,0,0010h

BIcoloff equ      20

transBI  clr.l    d3      lea      BIT,a2      ; for i = 0 to n-1

tBI2     clr.l    d2      ; for j = 0 to m-1

tBI3     mulu     move.w  d2,d4      move.w  d2,d4      ; get BI(i,j)
         lea      BI,a1      #BIcoloff,d4
         adda.l   d4,a1
         move.l   d3,d4
         lsl.l    #2,d4
         move.l   0(a1,d4),(a2)+ ; store at BIT(j,i)

         add.w    #1,d2      ; next j
         cmp.w    #4,d2
         bne      tBI3

         add.w    #1,d3      ; next i
         cmp.w    #5,d3
         bne      tBI2

```

| | rts | page | |
|--------|-------------------------------------------------------------------------------------|----------------------------------------------------------------------|------------------------------------------------|
| neg7 | move.l fpwait fpcomm move.l fpwait rts | #441ah d7,operand #6400h operand,d7 | ; -d7 -> fp0 ; fp0 -> d7 |
| div70 | move.l fpwait fpcomm move.l fpwait fpcomm move.l fpwait rts | #4400h d0,operand #4420h d7,operand #6400h operand,d0 | ; d0 -> fp0 ; fp0 = fp0/d7 ; fp0 -> d0 |
| mult05 | move.l fpwait fpcomm move.l fpwait fpcomm move.l fpwait rts | #4400h d5,operand #4423h d0,operand #6400h operand,d5 | ; d5 -> fp0 ; fp0 = fp0 * d0 ; fp0 -> d5 |
| add52 | move.l fpwait fpcomm move.l fpwait fpcomm move.l fpwait rts | #4400h d2,operand #4422h d5,operand #6400h operand,d2 | ; d2 -> fp0 ; fp0 = fp0 + d5 ; fp0 -> d2 |
| sub52 | | #4400h | ; d2 -> fp0 |


```

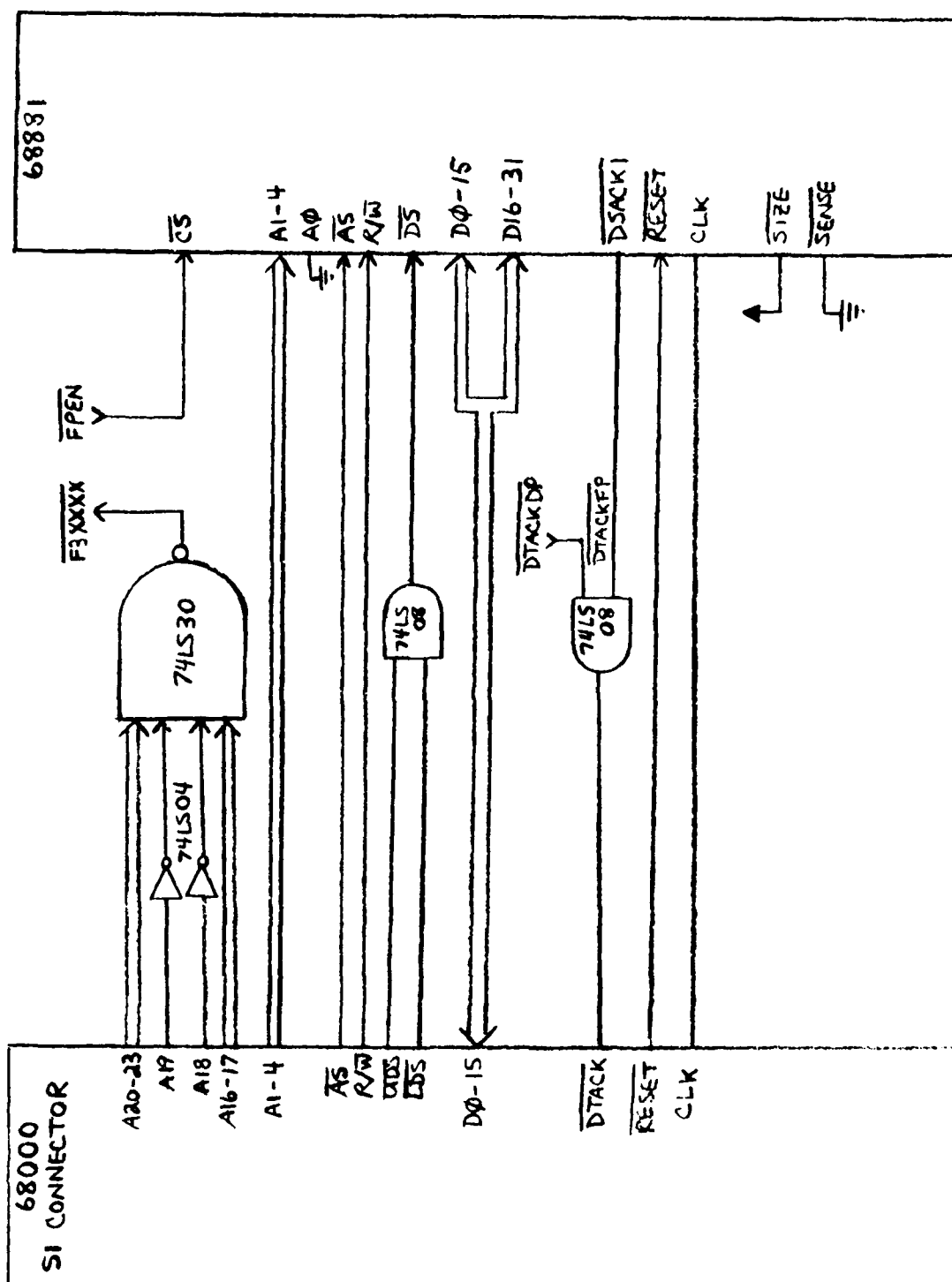
move.l d2,operand
fpwait
fpcmm #4428h
move.l d5,operand
fpwait
fpcmm #6400h
move.l operand,d2
fpwait
rts

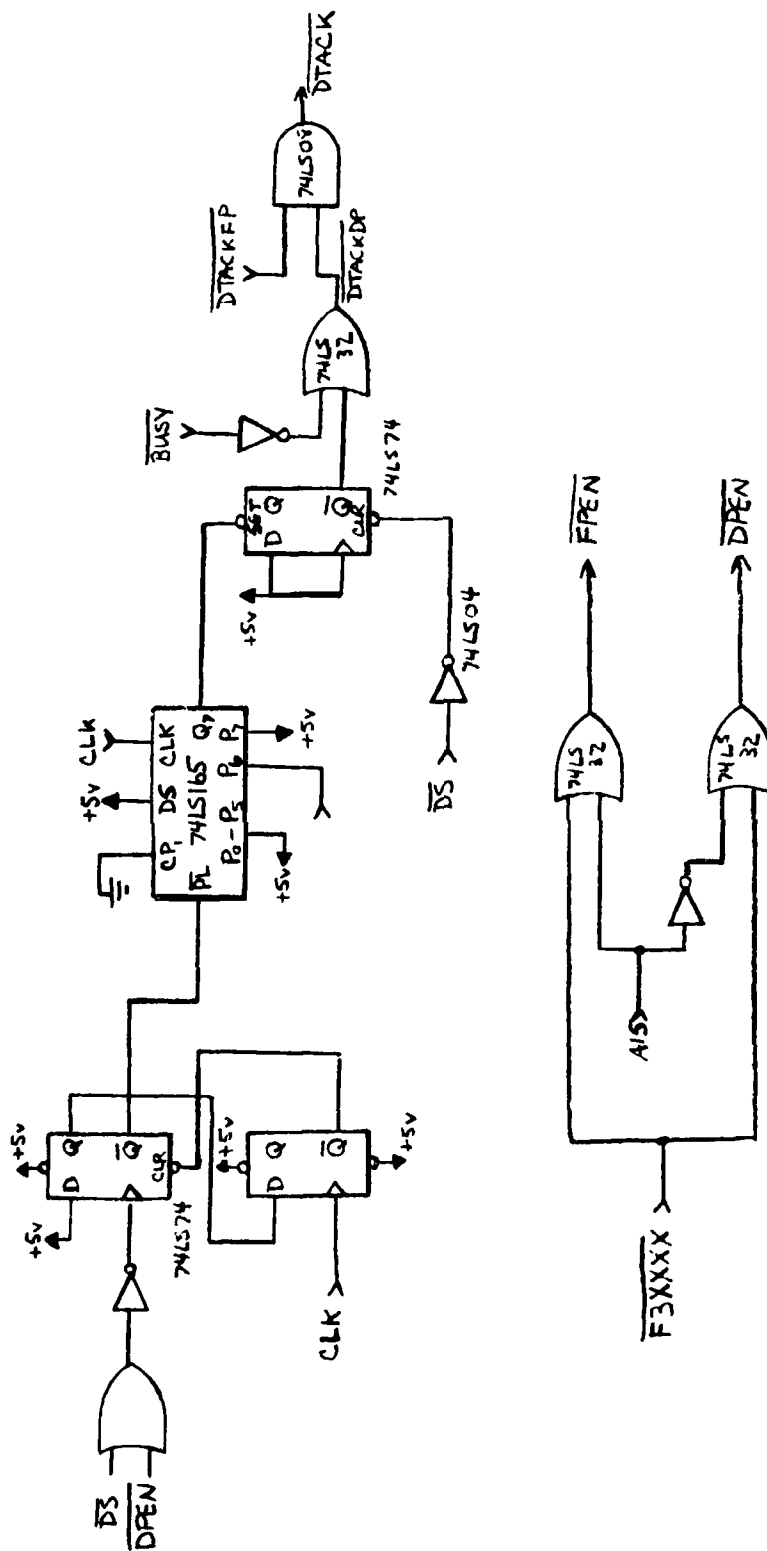
mult65
fpcmm #4400h
d6,operand ; d6 -> fp0
move.l fpwait
fpcmm #4423h
move.l d5,operand ; fp0 = fp0 * d5
fpwait
fpcmm #6400h
move.l operand,d5 ; fp0 -> d5
fpwait
rts

neg6
fpcmm #441ah
d6,operand ; -d6 -> fp0
move.l fpwait
fpcmm #6400h
move.l operand,d6 ; fp0 -> d6
fpwait
rts

```

Appendix A4
Schematics





AD-A194 886

A MULTIPROCESSOR AVIONICS SYSTEM FOR AN UNMANNED
RESEARCH VEHICLE(U) AIR FORCE WRIGHT AERONAUTICAL LABS
WRIGHT-PATTERSON AFB OH D B THOMPSON MAR 88

3/1

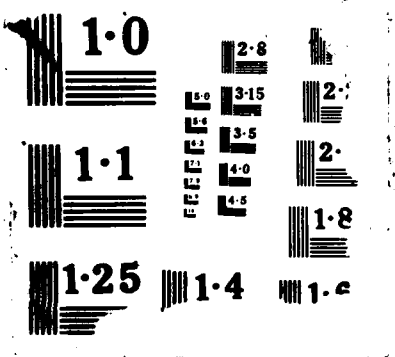
UNCLASSIFIED

AFWAL-TR-88-1003

F/G 12/6

NL





REFERENCES

- [1] D. Bursky, "32 bit Microprocessors", Electronic Design, January 8, 1987
- [2] R. Johnson, "Finally, Intel's 386: 4 MIPS And Mountains Of Memory", Electronic Engineering Times, October 14, 1985
- [3] R. Cushman, "EDN's 14th Annual Microprocessor/Microcontroller Chip Directory", EDN, November 26, 1987
- [4] S. Larimer et al, "A Continuously Reconfiguring Multi-Microprocessor Flight Control System", AFWAL-TR-81-3070, AFWAL/FIGL, May 1981
- [5] D. Thompson et al, "AF Multiprocessor Flight Control Architecture Developments: CRMMFCS and Beyond", NAECON 86 Proceedings, May 1986
- [6] K. Driscoll et al, "Multi-Microprocessor Flight Control System", AFWAL-TR-84-3076, AFWAL/FIGL, September 1984
- [7] M. Rosenbaum, "Survey of High Performance Parallel Architectures", BDM/ROS-86-1246-TR, BDM Corp, December 23, 1986
- [8] E. Meyer, "Survey of Multiprocessors", VLSI Systems Design, November 1985
- [9] D. Davis, "Parallel Computers Diverge", High Technology, February 1987
- [10] Ford EEC Subsystems and Applications Department, "Ford 8061 Software Manual", Ford Motor Company, July 1982

REFERENCES (CONTINUED)

- [11] R. Brietzman et al, "Development of an Optimal Automotive Control Custom Microprocessor", SAE Technical Paper Series, 820250, February 1982
- [12] D. Lieberman, "Electronic Products Forum: High Performance Busses", Electronic Products, June 16, 1986
- [13] J. Beaston et al, "32-bit buses - the silicon interface question", Integrated Circuits Magazine, February 1985
- [14] K. Marrin, "Bus differences more significant in principle than in practice", Computer Design, November 1, 1985
- [15] Mizar Inc, "MZ8115 CPU Module Users Manual", Revision E, November 1986
- [16] K. Marrin, "Real Time Operating Systems Prove Difficult To Evaluate", EDN, July 11, 1985
- [17] Hunter and Ready Inc, "VRTX/68000 Users Guide", April 1986
- [18] Industrial Programming Inc, "MTOS-68K Users Guide", March 1984
- [19] Motorola Inc, "MC68000-Family Real Time Multitasking Software Users Manual", October 1983
- [20] Software Components Group, "pSOS-68K Real Time Multitasking Operating System Kernel Users Manual", December 1986
- [21] Software Components Group, "pRISM-68K Real Time Interprocessor System Manager Users Manual", February 1987
- [22] Hunter and Ready Inc, "Dijkstra Semaphores Application Note", August 1983
- [23] M. Ben-Ari, "Principles of Concurrent Programming", 1982, Prentice-Hall

REFERENCES (CONTINUED)

- [24] Hunter and Ready Inc, "VRTX and Event Flags Application Note", August 1986
- [25] Hunter and Ready Inc, "Multi-Processor Applications Using VRTX Application Note", May 1984
- [26] Motorola Inc, "MC68881 Floating Point Coprocessor Users Manual", 1985
- [27] S. Harris et al, "Software Links Math Chip to 68000-Family Microprocessors", EDN, January 23, 1986
- [28] K. Rattan, "Study of Control Mixer Concept for Reconfigurable Flight Control System", Final Report, AFOSR, September 21, 1984
- [29] W. Press, "Numerical Recipes", 1986, Cambridge Press
- [30] G. Strang, "Linear Algebra and its Applications", 1980, Academic Press
- [31] G. Andrews et al, "Concepts and Notations for Concurrent Programming", ACM Computing Surveys, March 1983
- [32] Motorola Inc, "VME Bus Specification Manual", Revision B, August 1982
- [33] B. Ray, "Aerodynamic Characteristics of the XBQM-106 Unmanned Research Vehicle Modified For Flight Control Reconfiguration", AFWAL-TM-86-183, AFWAL/FIGC, April 1986

VITA

Daniel B. Thompson was born on 12 October, 1960 in Dayton, Ohio. He graduated from Wayne High School in 1978 and attended Wright State University in the Computer Engineering program. During his studies at Wright State University, he participated in the cooperative education program by working at the Air Force Flight Dynamics Laboratory. His primary duties involved designing and testing the software used on the Continuously Reconfiguring Multi-Microprocessor Flight Control System (CRMMFCS) laboratory demonstrator. He graduated from Wright State in December 1982 and accepted a position with the Flight Dynamics Laboratory. Since this time, he has been researching the areas of fault tolerance and parallel processing as applied to flight control and vehicle management systems. He currently is the program manager for the Advanced Multiprocessor Control Architecture Definition (AMCAD) in-house program. In September 1983, he began the Master's program in Computer Engineering at Wright State.

END

DATED

FILM

8-88

Dtic